LEVEL

MIT/LCS/TR-224

DATA-STRUCTURING OPERATIONS IN CONCURRENT COMPUTATIONS

DTIC
ELECTE
APR 18 1980
C

David Lee Isaman

DATA-STRUCTURING OPERATIONS IN CONCURRENT COMPUTATIONS

by

David Lee Isaman

Doctoral Thesis

© David Lee Isaman 1979

MIT/LCS/TR-224

October 1979

Massachusetts Institute of Technology

Laboratory for Computer Science

Cambridge, Massachusetts

02139

DATA-STRUCTURING OPERATIONS IN CONCURRENT COMPUTATIONS

by

DAVID LEE ISAMAN

Submitted to the Department of Electrical Engineering and Computer Science on July 23, 1979 in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

ABSTRACT

This thesis proposes operational specifications for a Structure Memory (SM). A specialized hardware component of a general-purpose computing system, the SM would directly execute operations on dynamically-structured data stored in it. The computing system is assumed capable of exploiting program concurrency at the machine-instruction level. For explanatory purposes, the proposed structure operations are presented in the context of the data flow model of concurrent computation.

Concurrency among a set of program instructions which all examine or modify the same structure must be carefully controlled, if the program is to be determinate. The first of two major contributions of the thesis is a combination hardware/software discipline which affords maximal concurrency consistent with determinacy. Its key feature is that the SM will not return a given pointer until certain previously-returned pointers to the same structure are no longer available as operands.

The second major contribution is the entry-execution model of concurrent computation. Reversing the emphasis of most previous work, this model concentrates on the operations performed by instructions, while abstracting away details of how operands are passed among them and how their execution order is determined. The essence of structure operators, that the result of an execution of one may depend on the input to previous executions of that and other operators, is given a natural expression in the new model. A proof of sufficient conditions for determinacy of a program containing structure operators is made more generally applicable through use of the entry-execution model as its medium.

ACKNOWLEDGEMENTS

The continued support of Professor Jack B. Dennis through the long and complicated course of this thesis is gratefully acknowledged. His efforts as thesis supervisor contributed substantially to the quality and completeness of the final product. Thanks are also extended to Clement Leung, Ken Weng, and Professor Barbara Liskov for their thoughtful comments on early drafts. The long-distance cooperation of Marilyn Pierce of the Department Graduate Office was invaluable.

Special appreciation goes to two people. The early-graduate-career mentorship of Professor Michael Dertouzos fostered a strong professional self-confidence, which has been a great source of internal resolve. The love and support of Janet McAfee has made the work of completing this thesis much more bearable.

Accession For	
NTIS &..&I	✔
DDC ...	
Un......	
J........	
By	
D	
Dist	Avail ... or special
A	

DEDICATION

To all those whose caring

showed me there is more to life

than writing a thesis.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

This thesis proposes specifications for a Structure Memory (SM).
A specialized hardware component of a general-purpose computing system,
the SM would directly execute operations on data structures stored in it.
The software overhead incurred in molding complex structures to conform
to the elementary organization of a conventional random-access memory
would thereby be greatly reduced.

Brief consideration of possible SM implementations suggests a
potential for executing several operations concurrently (during the
same time interval). Exploitation of this ability could result in
enhanced SM performance. Unfortunately, concurrent operations on data
structures can cause different runs of a program on the same input to
produce different outputs. Therefore, techniques must be found for
controlling any potential SM concurrency to prevent this intolerable
unpredictability. The first of two major results of the thesis is a
combination hardware/software discipline making it easy to eliminate
all dangerous concurrency at the sacrifice of little safe and productive
concurrency.

The second, possibly more significant result is the entry-execution
model. This radically-different model of concurrent programming reverses
the emphasis of existing models, concentrating on the operations performed
by instructions, while abstracting away details of how operands are passed
among them and how their execution order is determined. The generality of

the correctness proof for the new concurrency-control discipline testifies to this model's usefulness.

Section 1.1 below presents the argument for a hardware SM. It shows how the SM may be able to support concurrent operations and how this may have undesirable consequences. Section 1.2 presents a chapter-by-chapter overview of the logical progression of steps taken in ths thesis. Section 1.3 concludes with a brief survey of related work.

1.1 Motivation

An important part of many computer programs is the manipulation of data structures. One way of viewing a data structure is as a set of ordered pairs (s,e), in which s is a _selector_ and e is an _element_ of the structure. A selector is an atomic datum (typically either an integer or a character string) which serves to distinguish its associated element from all others in the structure; therefore, no two ordered pairs in one structure may contain the same selector. An element is either an atomic datum or another data structure.

Figure 1.1-1 portrays the simplest type of data structure, a one-dimensional array, in a graphical representation. All elements of an array are atomic data of the same type (e.g., integer, real, character). The selectors form a consecutive sequence of integers; in the example, those from 1 through 4. Each element is depicted as a node at the lower level of the graph in the Figure; a node representing an atomic element has that atom written inside it. The single node at the upper level of the graph represents the structure as a whole. For each ordered pair (s,e) in the structure, a branch labelled with s is drawn from the node
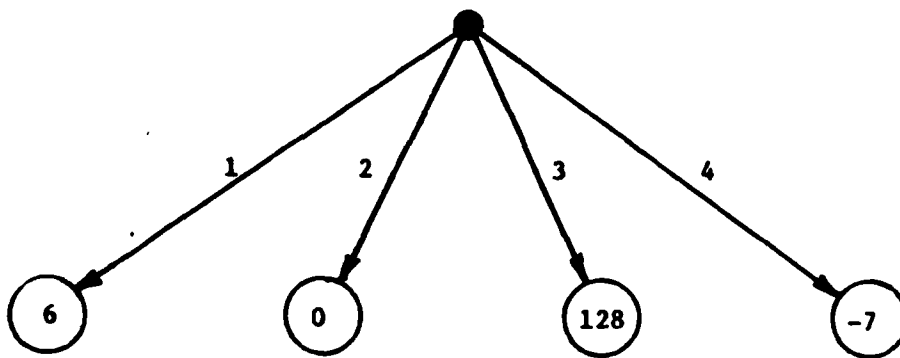
A One-Dimensional Array

Figure 1.1-1
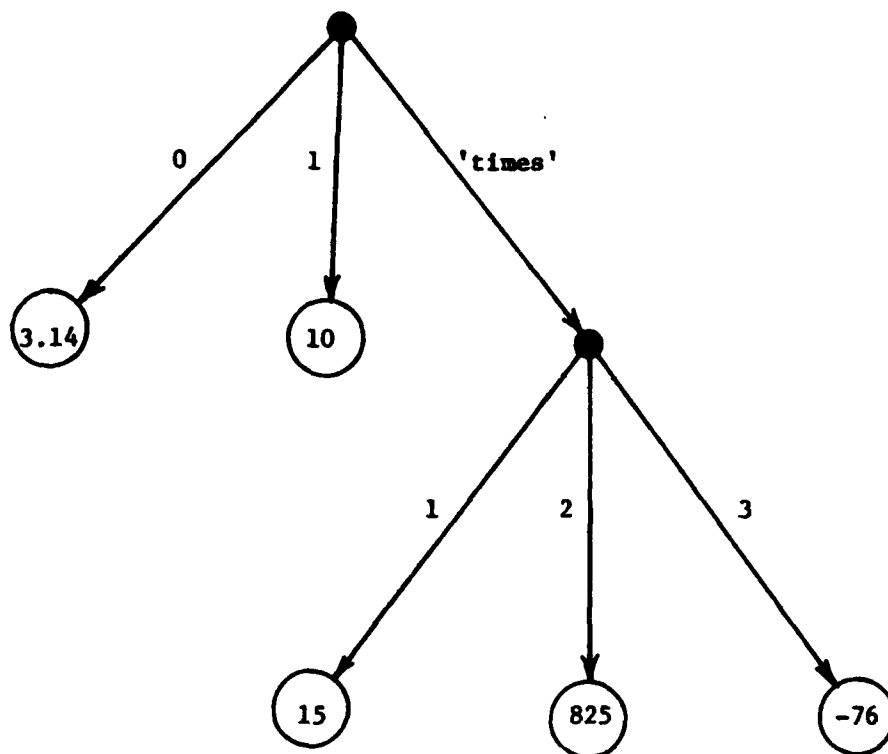
A More General Structure

Figure 1.1-2

representing the whole structure to the node representing the element e.

Figure 1.1-2 displays a more complex data structure. The elements of this are of different types: an integer, a real, and another structure, an array of three elements. Representing every structure as a separate node has allowed a consistent graphical treatment of both atoms and structures as elements. This second example also has more general selectors, encompassing character strings as well as integers.

The most primitive operation on a data structure is to determine its elements. This operation, called Select, takes a structure and a selector, and returns the element paired with that selector in the structure. With data structures defined as above, however, a single, general Select operation may pose a hazard: The element returned may be input to other Select operations if it is a structure, but not if it is an atom; conversely, it may be input to data-processing operations (such as arithmetic) if it is an atom, but not if it is a structure. Of the several alternative methods of eliminating this hazard, the one chosen is to redefine a data structure, based on its graphical representation. Atomic elements and structures alike are depicted as nodes. Each node has either (1) an atom associated with it, or (2) labelled branches emanating from it. A data structure is correspondingly redefined to be a set containing (1) an atom, (2) some selector-element pairs, where an element now is always a structure, or (3) both. A structure according to the old definition is made a structure according to the new definition by replacing each atom with the set containing that atom.

With this revised concept of data structure, a Select operation can take any structure and any selector, and always return a structure.

If the input structure has no ordered pair containing the input selector, an exception (similar to an arithmetic overflow) occurs, whether or not the structure contained only an atom. A second operation, Fetch, retrieves the atom in a structure; applying it to a structure which has no atom results in an exception. Structure-altering operations include Assign, which replaces the atom in a structure (or adds one if there was none).

The set of operations Select, Fetch, and Assign is sufficient to handle _static_ data structures. A static structure is one whose graph representation can never change shape; only the atoms inside the nodes can be altered. Frequently, however, it is desirable that an unpredictable amount of input data be retained in a structured form. This requires the ability to manage _dynamic_ data structures, the shape of whose graphs may be altered by the addition and deletion of nodes and branches. A prime example of this is the symbol table in a programming-language processor (compiler or interpreter). Each element e of a symbol table is a structure describing the linguistic attributes of one symbol T in the program being processed; the selector paired with e is most conveniently the symbol T itself. Neither the elements nor the selectors in the symbol table are known when the language processor starts; hence the need for operations to create new structures from existing structures and new selectors.

The first encounter with each new symbol in the program triggers a sequence of steps to add it to the symbol table. Figure 1.1-3 pictures what the last step might be. Part (a) shows schematically the existing symbol table and the smaller structure constructed to describe the new symbol. The final step is the creation, in a single operation, of the new symbol table in part (b). (Whether or not the old symbol table
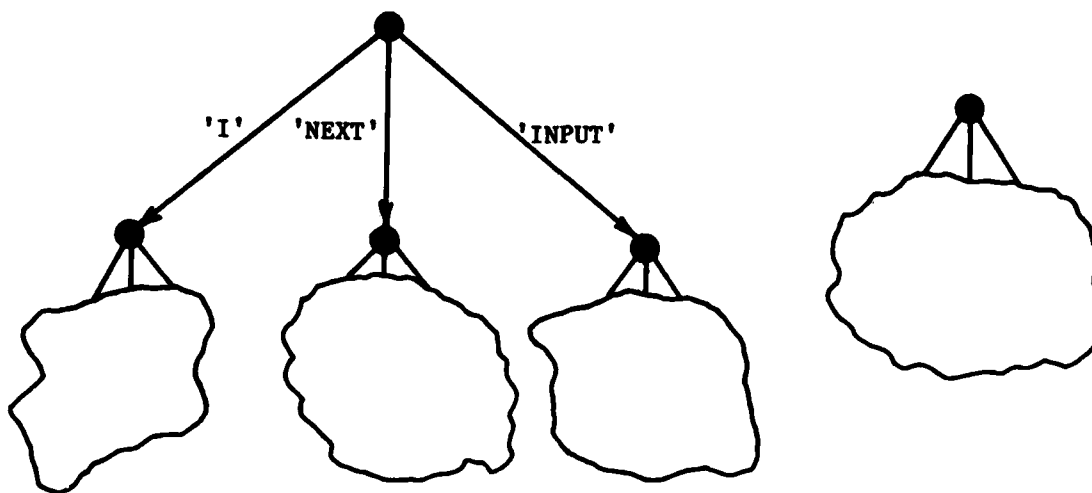
Before

(a)
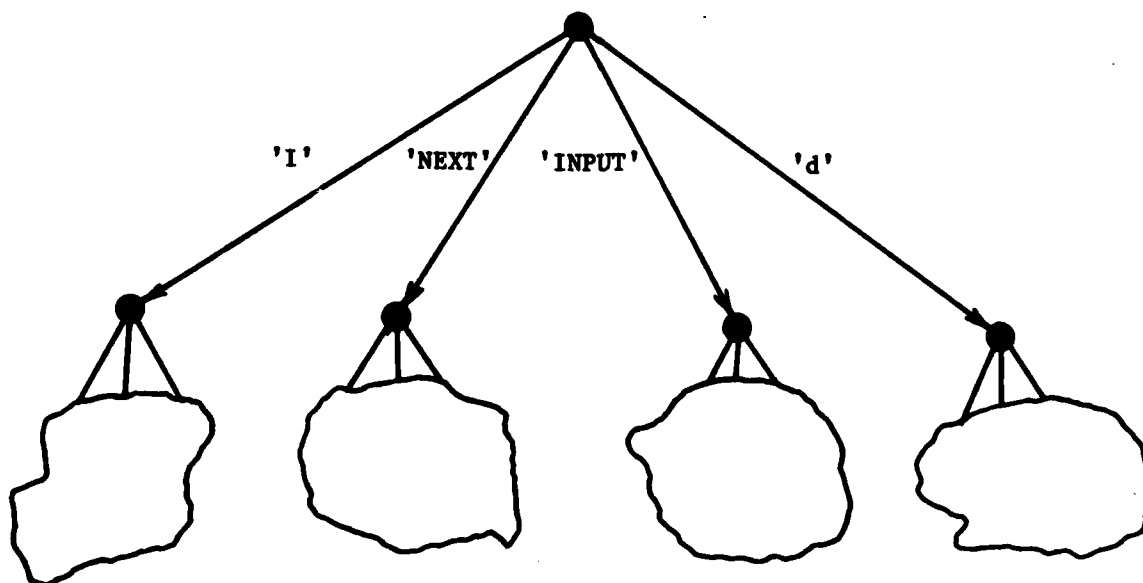
After

(b)

Combining Two Dynamic Data Structures

Figure 1.1-3

continues to exist apart from the new one is a key issue of the thesis.
The subtle distinction will be discussed later in this chapter.) Also
useful are an operation to create a new structure by removing an element
from an existing one and operations to enumerate all of the selectors in
a structure.

The only memory technology in which it currently is feasible to store
large amounts of information is the random-access memory (RAM). A RAM is
organized as a one-dimensional (or possibly two-dimensional) array, a
homogeneous collection of storage cells, each capable of retaining one
atom, with selectors (addresses) which form a sequence of consecutive
integers. RAM hardware can support static data structures; that is, it
can directly execute operations analogous to Select, Fetch, and Assign,
with the same speed with which it accesses any stored atom. Support of
dynamic data structures, however, necessitates elaborate software systems
which are expensive, first to write and verify, and then to execute.

Great bodies of literature have grown up concerning software
approaches to the two major aspects of implementing dynamic data
structures:

1. Storage allocation - deciding which set of physical cells shall
   store each newly-created structure, and deciding when those cells
   can be re-used because the structure they store can no longer be an
   operand to any operation.

2. Searching - storing the set of selectors in a structure so that the
   Select operation can be performed quickly (e.g., in a hash table).

Even with the most well-written software, dynamic data-structuring
operations are several times slower than basic RAM accesses.

These observations inspire consideration of the possibilities and
potentialities of shifting the support of dynamic data structures from
software to a hardware Structure Memory (SM). The SM would serve as an
adjunct to the RAM in a computing system, as diagrammed in Figure 1.1-4.
The RAM continues to store programs, non-structured data, and static
structures, while the SM directly executes operations on dynamic struc-
tures. The Control Unit (CU) fetches and decodes instructions from the
RAM, finds their operands, and sends these to the appropriate system
component: the RAM, the SM, or a functional unit (for data-processing
and I/O operations).
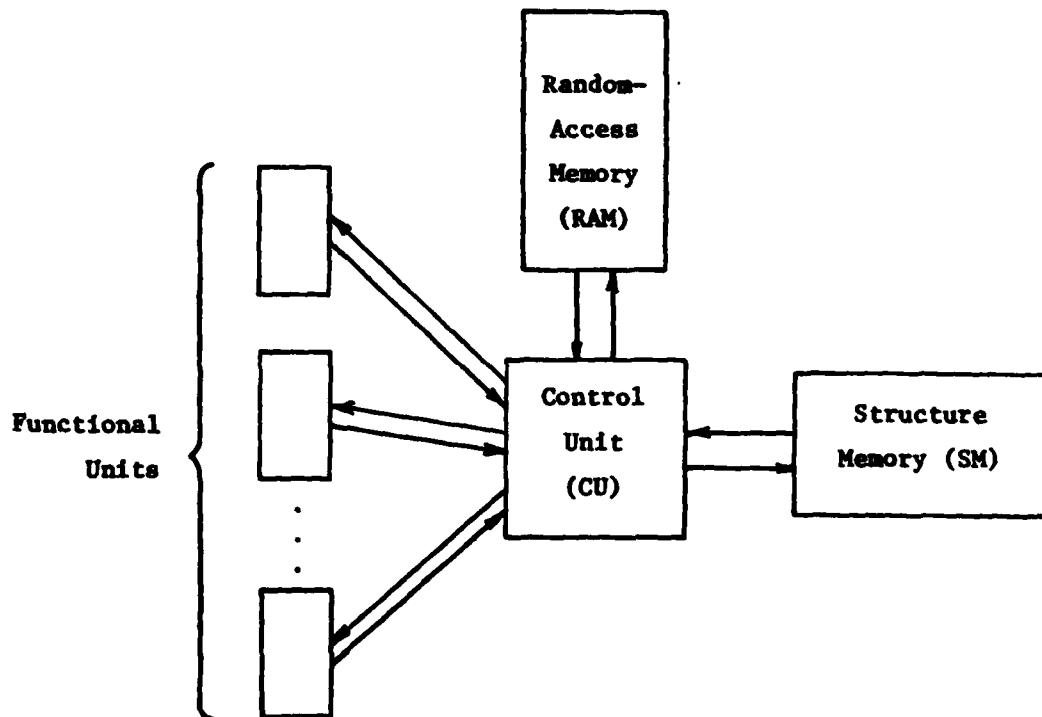
Proposed Computer System Organization

Figure 1.1-4

There are ample precedents for this development. The performance of many small mini-computers can be improved by shifting the execution of floating-point operations from software routines into a hardware functional unit [26]. The importance of the stack, a simple dynamic structure, has prompted the inclusion in many instruction sets of special push and pop operations, which replace sequences of two or three conventional instructions [36]. In the SYMBOL machine, strings are manipulated directly by a separate unit called the Memory Controller [29]. There are machine instructions to fetch an addressed group of eight bytes and return the address of the following (or preceding) group in the string, and to append or insert a group into a string. A structure may be formed by storing the address of one string in another string. String storage space is allocated automatically (by hardware) as it is needed, and a single instruction will deallocate all storage occupied by a string, even if it is structured. The SM envisioned here would extend the concept of this Memory Controller to encompass the manipulation of structures with selectors (so that a program need not fetch and search an entire structure to find a given element).

To minimize the amount of information which must be moved between the CU and the SM, all dynamic structures will be stored within the SM. References to stored structures will be communicated outside the SM by means of pointers. A pointer is an arbitrary bit string which is associated by the SM with a unique structure stored therein. Pointers have no intrinsic meaning outside the SM; therefore, it is important that other units of the computing system merely pass pointers around, never attempting to perform operations (e.g., arithmetic) on them.

The implementation of an SM would probably store a structure by physically associating with its pointer a _content_. The content for the structure $\{v,(s_1,e_1),\ldots,(s_n,e_n)\}$ consists of a bit string encoding of the atom v and the ordered pairs $(s_1,p_1),\ldots,(s_n,p_n)$, where p_i is the pointer to e_i, $i=1,\ldots,n$. The entire content would be stored in physically-adjacent locations, to minimize the effort required to search it for a given selector. This implies that a content may have to be moved to another set of physical locations, if it or another stored content changes size (through an operation such as that illustrated in Figure 1.1-3), or if the storage space becomes fragmented [10]. Therefore, the pointer associated with a structure cannot be treated simply as an unchanging physical address of the structure's content. A mobile content can be located only if there is a _key_ stored within it (or adjacent to it) by which it can always be recognized; the obvious choice for a key is the pointer to the structure for which this is the content. The SM therefore must contain an _associative memory_, one which can compare a given key (the _search key_) against all stored keys and return the locations(s) of any matching key(s). The associative memory would return the location storing that key; the content of the structure is then known to occupy locations adjacent to that.

There are two basic techniques for implementing an associative memory: parallel and serial. In the former, all stored keys are compared against the search key at the same time; in the latter, the stored keys are compared one at a time. The parallel method results in the fastest access time, but requires much more hardware: one comparator circuit per stored key. The amount of comparator hardware needed by the serial

technique is negligible, but finding a match may entail making a complete pass through the memory, comparing against every stored key.

The performance of the inexpensive serial associative memory can be greatly improved by adding comparators dedicated not to different stored keys, as in the fully-parallel approach, but to different search keys [18]. Then as the stored keys are retrieved, one-by-one, from the serial memory, each can be compared to several search keys simultaneously. The guaranteed number of matches per complete pass through the memory is thus increased from one to the number of search keys available at the start of the pass.

The search keys given to the associative memory in the SM are pointer operands of structure operations. Exploiting the ability to search for several keys at once requires the following: the operands of executions of several operations can be sent out from the CU to the SM without the results of any of the executions having been returned; in this case, those operations are defined to be concurrent. The simplest example of concurrent operations is seen in the evaluation of the expression (a+b)*(c+d). The operands of the multiplication cannot be sent out until the results of the two additions have been returned; therefore, the multiplication is not concurrent with either of the other operations. However, the operands of both additions can be sent out before the results of either have been returned; i.e., the two additions are concurrent.

There are at least three possible reasons for which concurrent operations may be desirable or necessary: In a very slow device such as a serial associative memory, the total time required to execute a set of concurrent operations can be reduced by a factor as large as the size of the set. While a reduction this large is by no means assured, total

execution time does generally decrease as the number of concurrent oper-
ations increases. Secondly, a fully-parallel associative memory might
provide structure operations which are as fast as any other operations, but
between executions, this expensive device would sit idle. As the number of
concurrent operations increases, the number of operations executed per unit
time increases; this greater utilization can be a strong economic incentive
for concurrency. Finally, there are applications which are inherently
concurrent, such as real-time systems responding to external events. Since
these events can occur in any order, the operations they invoke must be
able to execute in any order.

These arguments for concurrency are offset in the case of structure
operations by a peculiar danger: concurrency may compromise a program's
functionality. A functional program is one which, every time it is run on
the same inputs, produces the same outputs. The simplest example of the
danger is the case of a Fetch and an Assign operation which necessarily
operate on the same structure. If these are concurrent operations, then
the order in which they are executed by the SM is not fixed. If the Fetch
is executed first, it returns the atom in the structure's original content;
if it is executed second, it returns the new atom stored by the Assign
execution. Thus, in two runs of a program on the same input, the same
execution may have a different result, which may in turn lead to different
program outputs. I.e., concurrent operations on the same structure may
cause a program to be non-functional.

The goal of the thesis is to specify a Structure Memory which supports
concurrent operations in such a way that it is easy to guarantee that they
do not induce non-functionality. Nothing more will be offered on the

subject of implementing the SM; the only concern is for specifying an operation's results returned from the SM as a function of the preceding sequence of operands sent to the SM. The primary design criterion is to maximize the allowable concurrency of structure operations, consistent with functionality. The secondary criterion is to minimize the computational complexity of distinguishing functional from non-functional programs, if both are possible.

It is assumed that the SM is used in conjunction with a CU in which any two structure operations which could be concurrent are concurrent. The capabilities of a CU *vis-à-vis* concurrent operations are expressed abstractly by a <u>model</u> <u>of</u> <u>concurrent</u> (or parallel) <u>computation</u>, consisting usually of two components:

1. A parallel-programming language, a collection of programs each consisting of (a) a set of instructions, (b) a diagram of where an instruction gets its operands and where its results go (the <u>data</u> <u>flow</u>), and (c) a diagram of which instructions' results must be returned before which other instructions' operands can be sent (the <u>control</u> <u>flow</u>).

2. A method for generating descriptions, to some level of detail, of the possible behaviors of the CU (<u>computations</u>) when given any program in the language and any input to that program.

Although for clarity, results are derived using a specific model, it is desired that the SM specification be expressed as abstractly as possible, i.e., divorced from any particular model of concurrent computation.

1.2 Plan of the Thesis

Chapter 2 contains a precise statement of the goal of the thesis and of the approach taken to achieving that goal. It also formally introduces the data flow model of concurrent computation (so called because the same diagram which shows a program's data flow also specifies its control flow). Data flow was chosen as the concrete model in which to derive results for three reasons: (1) it provides the simplest and most natural expression of concurrency, (2) all data-flow programs having no structure operations are automatically functional [12], and (3) there are several efforts under-way to implement a Control Unit based on the data flow model [4, 8, 15, 20, 33, 35].

The mechanism for generating computations is a non-deterministic automaton called an interpreter. The interpreter is defined by (1) a set of possible states, and (2) a non-deterministic state-transition rule, specifying how any state is transformed into any of one or more possible next states. A program P together with an input to P establishes an initial interpreter state. The computations are the possible state sequences generated from this initial state by successive applications of the state-transition rule. Every final state of one of these sequences describes a program output for P; if P is non-functional, then different state sequences starting in the same initial state may lead to different final states.

Several data-flow languages and interpreters are developed in the thesis. The first language described is the basic data-flow language L_B. This is assumed to include an unspecified complement of operations on

atomic data, as well as control constructs for conditional branching and looping. The two languages L_{BV} and L_{BS} are then formed by augmenting L_B with two similar systems of data-structuring operations: the Structure-as-Value (S-V) and the Structure-as-Storage (S-S) systems. Computations are generated from programs in all three languages by the single <u>standard data-flow interpreter</u>.

The S-V and S-S systems illustrate two approaches to achieving the goal of the thesis. The fundamental difference between them can be explained by reference to Figure 1.1-3. Part (a) of the Figure shows two structures which are to be "combined" into the single structure of part (b). The S-S system includes an operation (Update) which will change the stored content of the larger original structure m, adding an ordered pair consisting of the selector 'd' and the pointer to the smaller structure. Thus the same pointer (to m) points to different structures at different times. The S-V system, on the other hand, contains no operation which can change the content of a structure after a pointer to the structure has been returned from the SM. Instead, the S-V operation Append accomplishes the effect pictured in Figure 1.1-3 by creating a distinct new structure n, whose stored content equals m's content with the new selector-pointer pair added. Once a pointer to n is returned from the SM, it always points to a structure identical to that in Figure 1.1-3(b), while the pointer to m continues to point to a structure identical to the larger one in Figure 1.1-3(a).

The difference between the S-V and S-S systems with regard to the goal of the thesis can be stated succinctly: Given two programs, one in L_{BV} and one in L_{BS}, to do the same thing, the L_{BS} program may have more

concurrency, but it also may be non-functional; the L_{BV} program is necessarily functional. I.e., L_{BV} solves the problem of guaranteeing functionality, but at the cost of some potential concurrency; L_{BS} recovers this loss, but in so doing permits additional concurrency which may induce non-functionality. This inspires the search (in Chapter 3) for a third language-interpreter combination which eliminates from L_{BS} just the dangerous concurrency. For every L_{BV} program P (which is necessarily functional), there is a functional program P' in the new language which is equivalent to P (i.e., which produces the same outputs given the same inputs); furthermore, P' contains much (if not all) of the safe concurrency missing from P. Formal definitions of a functional data-flow program with data structures and of equivalence between two such programs are given at the end of Chapter 2.

Chapter 3 commences with a study of the cause of non-functionality in L_{BS} programs on the standard interpreter: conflict. Certain pairs of concurrent operations (e.g., a Fetch and an Assign) conflict if it is possible that equal pointer operands to executions of the operations are in the SM at the same time. The easiest way to guarantee functionality is to eliminate all possible conflicts. This results in a determinate program, one in which each execution always has the same operands and always produces the same results in all computations on a given program input.

A novel new two-pronged technique for insuring freedom from conflict in an L_{BS} is then developed. First, each program is rewritten to satisfy the Determinacy Condition and the Read-Only Condition; the subset of all L_{BS} programs satisfying both is denoted L_D. Secondly, the interpreter is modified to delay return of the pointer result of a Select execution until

certain previously-returned pointers to the same structure have been input
by other executions. It is argued that every L_D program, running on the
modified interpreter, is functional; the remainder of the thesis is devoted
largely to a rigorous proof of this claim. The final section of Chapter 3
presents an algorithm to translate any L_{BV} program P into an L_D program P',
and proves that if P' is functional, then it is equivalent to P.

Chapter 4 introduces a radically-different model of concurrent compu-
tation, the entry-execution model. As noted earlier, existing models
concentrate on the data and control flow of a program, virtually ignoring
the actual operations performed by most of the instructions. The new model
ignores data and control flow, focusing instead on defining operations and
the effects of their concurrent execution. A computation in the entry-
execution model *consists of a sequence of* entries, the operand and result
values of executions, arranged in an order in which they might be sent from
and returned to the Control Unit. An algorithm is presented for con-
structing, from any data-flow language L and interpreter I, the entry-
execution model EE(L,I); this serves a dual role: as a concrete example of
such a model, and as the first step in applying the formal results to L_D
running on the modified interpreter M.

Chapter 5 develops a Structure-as-Storage (S-S) entry-execution model.
This demonstrates the principle (for which the entry-execution model is
particularly appropriate) of defining a set of operations by specifying
how the results of an execution depend on the preceding sequence of
executions' operands. This definition does not incorporate the concept of
a data structure; it is simply a description of the input/output behavior
expected of an SM which stores the structures and performs the operations

described earlier. Therefore, the Chapter also shows how to make the connection between the abstract model and concrete data structures which can be visualized.

Chapter 6 first defines determinacy in entry-execution terms. It then presents seven Determinacy Axioms, and proves quite generally that if any S-S model, of any concurrent computing system, satisfies these axioms, then it is determinate. Six of the axioms are standard: their importance to guaranteeing determinacy in systems without data structures (including data flow) has long been appreciated. The seventh axiom embodies the requirement for freedom from conflict between data-structuring operations.

Chapter 7 uses the result in Chapter 6 to prove that L_D running on the modified interpreter is functional. This is done in three steps: (1) verifying that $EE(L_D,M)$ is an S-S model, (2) proving that $EE(L_D,M)$ satisfies the seven Determinacy Axioms, and (3) showing that the algorithm by which $EE(L_D,M)$ was constructed could have produced a determinate model only if every L_D program running on M is functional. This leads to the final conclusion that the translation in Chapter 3 from L_{BV} to L_D does produce equivalent programs.

Chapter 8, the final chapter, summarizes the developments in the thesis, evaluates how well these meet the goals, and provides suggestions for further research.

1.3 Related Work

This section surveys past research on the topic of a Structure Memory
and the problem of guaranteeing functionality in the face of concurrent
structure operations. A characterization of existing models of concurrent
computation is provided as a prelude to the introduction of the entry-
execution model, in Section 4.1.

Gertz [18] studied the implementation of an SM using associative
memories. His Generalized Information Structure (GIS) was the same as that
defined here (Definition 2.2-1) (except that directed cycles are not
allowed). His choice of operations was unusual, however, because he was
primarily interested in storing data-flow-like programs in, and executing
them out of, the SM. The major results included:

1. the design of a system to execute, directly from the SM, GIS
    representations of parallel programs, including multiple concurrent
    activations of a single procedure, and

2. the development and analysis of stochastic models of modular,
    hierarchical SM's constructed of associative memories.

He did not directly address the issue of guaranteeing functionality.

Hawryszkiewycz [21] developed a scheme for coordinating concurrent
operations on a data base. He began by mapping relational data bases [7]
onto data structures like those being used here. He then gave a set of
semantic procedures, sequences of structure manipulations which implement
operations on relations. His primary correctness criterion, though a
little weaker, was not fundamentally different from the requirement of
functionality: The overlapped execution of two semantic procedures on the
structure representing a relation should result in the same transformation

as if one procedure (the first one invoked) had completed before the other
one had started. His solution to the coordination problem was very similar
to the one proposed herein (which was independently developed); it was,
however, less simple and general, for the following reasons:

1. His model of concurrent computation was based on present capabili-
    ties: sequential processes synchronized by semaphores. This necess-
    itated much attention to details (setting and testing locks, queuing
    up suspended processes) which tended to obscure the innovative
    mechanism. In the data-flow model, these effects are achieved much
    more easily.

2. He had additional criteria for correct coordination, requiring a
    mechanism more elaborate than would be necessary for simple
    functionality.

3. His coordination method was specialized to a particular set of
    semantic procedures; it is not clear how this would be generalized
    to arbitrary concurrent computations.

The Structure-as-Storage operations were introduced by Dennis in [11].
This paper recognized the danger of non-functionality and suggested (as in
the present work) eliminating it through a combination of program restric-
tions and interpreter modifications. The program restrictions (which were
extended in [17]) were essentially the Determinacy and Read-Only Condi-
tions. The interpreter modification, however, was far more extensive than
that proposed here: The standard interpreter passes data and permission
to execute from one program instruction to another, in one direction; the
modification of [11] required that permission to execute also be passed
back in the other direction, at least between structure operations.

Maximal concurrency is provided at the expense of greatly-increased execution time for structure operations by the interpreter modifications devised by Campbell-Grant [13]. His technique involved maintaining, for every pointer variable v, a list of all structures reachable from that pointed to by the current value of v. In general, every Select and Update execution will require changing one of these lists, incurring enormous overhead.

Both Rumbaugh [31] and Ackerman [1] offered sets of Structure-as-Value operations which were more complex than those in L_{BV}, due presumably to a stronger desire for programming ease and implementation efficiency. Both assumed that a structure's content consists of a fixed number of elements, each of which can be either an atom or a pointer; selectors were limited to consecutive integers. Select and Append operators could read atoms as well as pointers (eliminating the need for the Fetch and Assign operations). Ackerman provided just the Select and Append; Rumbaugh constructed more complex operations out of these. Both presented conceptual designs for at least part of the SM hardware (the reference-counting mechanism, explained in Section 8.2.1.1).

All of the Structure-as-Value systems (Rumbaugh's and Ackerman's, as well as the simpler L_{BV}) pay the same price for guaranteed functionality: the loss of structure concurrency (as explained at the end of Section 2.2.2, structure concurrency is the ability to read one sub-structure of a structure while another one is being changed). Only the modified Structure-as-Storage system presented in Chapter 3 will guarantee functionality while still allowing structure concurrency.

Chapter 2

Structure Operations and Concurrency

This thesis studies concurrent computations with two fundamentally-
different sets of structure operations: the Structure-as-Value (S-V)
operations and the Structure-as-Storage (S-S) operations. The most
striking differences between a program P using S-V operators and an
apparently-equivalent program P' using S-S operators are that:

1. P' exhibits more concurrency than P, but

2. P' might not be equivalent to P, because it might not be functional.

This chapter defines the two sets of structure operations, within the
framework of a specific model of concurrent computation called data flow.
Section 2.1 describes the basic data-flow language without structures, L_B.
Section 2.2 defines the languages L_{BV} and L_{BS} formed by augmenting L_B with
the S-V and S-S operations respectively. Section 2.3 illustrates, through
the use of examples, the two differences between programs in L_{BV} and L_{BS}.
Finally, Section 2.4 makes precise the primary goal of the thesis: To
develop a language, based on L_{BS}, in which, for every L_{BV} program P, there
is a program which is equivalent to P and maximally concurrent.

2.1 The Basic Data Flow Model

As explained in Chapter 1, a data-flow model consists of (1) a set of
programs, constructed according to certain syntactic rules, and (2) an
interpreter, which generates computations, as follows: A program together
with an input to it establishes an initial state of the interpreter; each
possible computation by that program on that input is an ensuing sequence

of interpreter states, generated according to a state-transition rule.

Section 2.1.1 below describes the syntax of a basic data-flow language L_B.

A single interpreter gives meaning to the programs in L_B, L_{BV}, and L_{BS};

this will be known here as the standard data-flow interpreter. That

portion of it pertinent to programs in L_B, which is adapted from the model

first described in [13], is defined in Section 2.1.2.

2.1.1 Data-Flow Programs

A data-flow program is a graph. The vertices of this graph represent

instructions, and the arcs represent local data storage.

Definition 2.1-1 A program in any data-flow language is a connected

directed graph over a set of labelled vertices called actors. The unique

label of each actor is drawn from an arbitrary but fixed set L. The

directed arcs terminating on an actor constitute the ordered set of input

arcs of that actor. The directed arcs emanating from an actor form the

unordered set of output arcs of that actor. No arc is an input arc of

more than one actor, and no arc is an output arc of more than one actor.

Those arcs which are not output arcs of any actor are the ordered set IN

of program input arcs; correspondingly, the ordered set OUT of program

output arcs comprises all those arcs which are not input arcs of any actor.

If IN contains m arcs and OUT contains n arcs, the program is an m,n data-

flow program.

Each arc in a program conveys one of two types of information — data or control — and consequently is known either as a <u>data</u> <u>arc</u> or a <u>control</u> <u>arc</u> (Figure 2.1-1). All data are drawn from an <u>atomic</u> <u>value</u> <u>domain</u> V for the language; control values are either <u>true</u> or <u>false</u>.

An input arc to an actor d stores a value until d uses it. At that time, the values on all of d's input arcs are removed, and used to compute results which are placed on d's output arcs. Thus d's results are available to just those other actors of which one of d's output arcs is an input arc.

A basic data-flow language has a minimal complement of control and data-processing actor types. From these can be constructed control structures corresponding to sequencing, conditionals, and iteration.

<u>Definition 2.1-2</u> A <u>basic</u> <u>data-flow</u> <u>language</u> is a data-flow language in which all actors are restricted to be from one of the following classes (illustrated in Figure 2.1-2):

1. atomic operator - An operator has an ordered set of $r>0$ input arcs, and two disjoint sets of output arcs: the number-1 group and the number-2 group. Either of these groups (but not both) may be empty.

data arc   control arc

Data-Flow Program Arcs

Figure 2.1-1

operators

T-gate F-gate merge gate

and or not
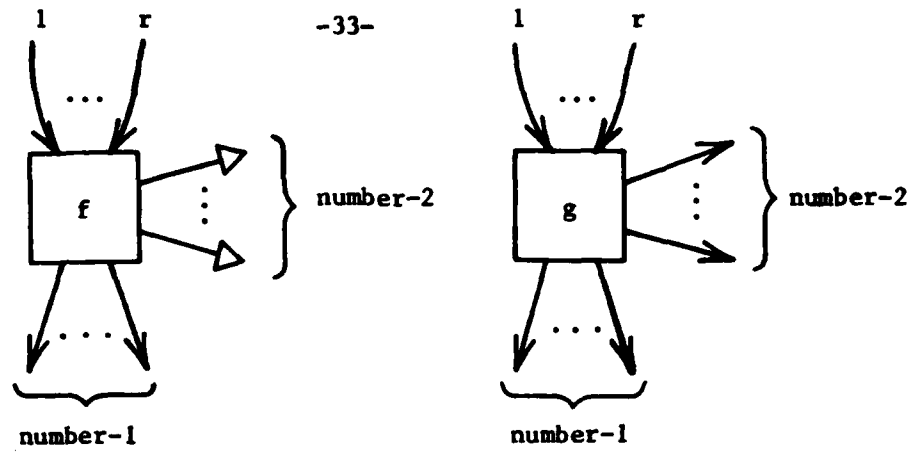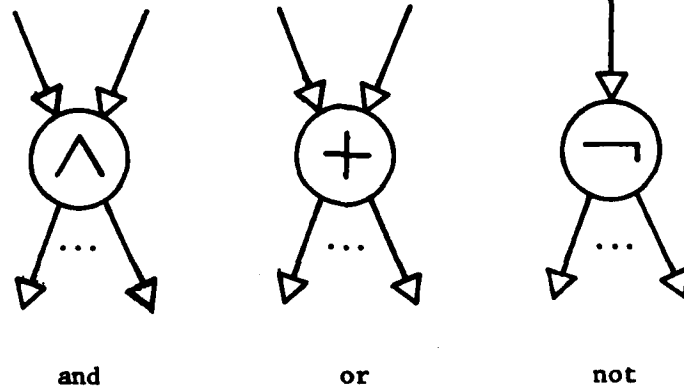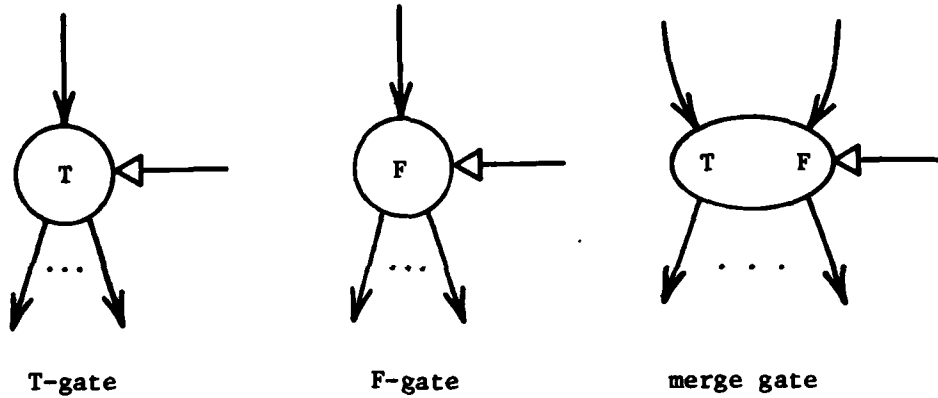
Actor Types in a Basic Data-Flow Language

Figure 2.1-2

All arcs in each group store one result of an execution of the actor, so that each execution produces one or two results. Usually, the number-1 group will be data arcs, and the number-2 group will be control arcs. This allows consistent treatment both of functions (which produce data values) and of predicates (which produce control values). These two output groups may be jointly defined, with a true control output signaling that the data output is meaningful; examples of the use of such hybrid operators will be seen later. Each r-input operator in a program has associated with it a total function:

$$V^r \rightarrow V \times \{\underline{true}, \underline{false}\} \quad \text{or} \quad V^r \rightarrow V \times V$$

Whenever the operator is executed, the values stored on its r-tuple of input arcs are combined to form the input r-tuple to the function. The values in the resulting output pair are then placed on the number-1 and number-2 output arcs, respectively.

2. T-gate and F-gate actors.

3. merge actors.

4. Boolean actors and, or, and not.

These last three types of actors control the flow of data through a program. Their operation is described in greater detail in the following subsection.

△

Figure 2.1-3 is a simple example of a program in a basic data-flow
language. It computes the value of the expression (a+b)*(c-d). The
four arcs in IN have been labelled a, b, c, and d to indicate the input
variable for which the arc represents storage. The 2-tuple of input arcs
of the non-commutative subtraction operator has been indexed.

Informally, a computation by this program proceeds according to
the following rules: At any time after the values of inputs a and b are
stored on their respective input arcs, the addition operator "fires". That
is, it removes a pair of values from its input arcs, applies its associated
function (addition) to this pair, and places the result on its output
arc. The subtraction operator acts similarly. Since none of the data
needed by a firing of one of these operators is produced by the other,
the operators are concurrent. Finally, when the outputs of both the
addition and subtraction are available, the multiplication operator is
enabled to fire.
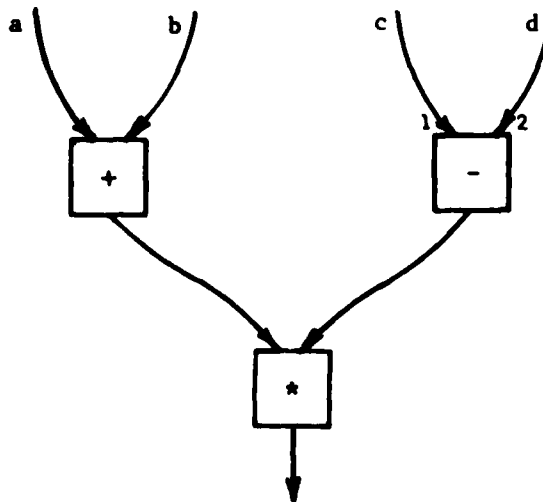
A Simple Data-Flow Program

Figure 2.1-3

A particular basic data-flow language is distinguished solely by its atomic value domain V and the functions available for operators. Typical would be the elementary data types and operations of standard programming languages: integer and floating-point arithmetic and string manipulations. The only element presupposed in this thesis is a distinctive value in V denoted by <u>undef</u>. A token with this value might be output as a result of, e.g., attempting to divide by zero. It is assumed that an otherwise arbitrary choice of V and of the data-processing functions has been made; the resultant basic data-flow language will be denoted L_B. The next subsection defines that portion of the standard interpreter's state and state-transition rule involved in interpreting programs from L_B.

2.1.2 State Transitions

A computation by a program P from L_B is a sequence of states of the standard data-flow interpreter. The only non-empty component of the standard interpreter state when interpreting P will be a <u>configuration</u> for P. This simply tells what arcs hold what values. Each transition from one state to the next in a sequence involves removing old values from some arcs and placing new values on other arcs; the graph containing these arcs, which is P, remains constant.

<u>Definition 2.1-3</u> A <u>configuration</u> of a data-flow program P from L_B is:

1. P, plus

2. an association of a value from V or the symbol <u>null</u> with each data arc of P, plus

3. an association of a symbol from the set {<u>true</u>, <u>false</u>, <u>null</u>} with each control arc of P.

△

Figure 2.1-4 is an example of an ALGOL program and a configuration of the equivalent 2,1 data-flow program. (This program computes the sum of the first N positive integers; its interpretation is explained shortly.) A solid circle is drawn on each arc with which is associated a non-null value, and a symbol denoting that value is written beside the circle. These circles are called <u>data</u> <u>tokens</u>, <u>true</u> <u>tokens</u>, and <u>false</u> <u>tokens</u>, according to the associated value. The figure depicts an <u>initial</u> <u>configuration</u> for the program: all program input arcs have tokens on them, as do certain control arcs.


```
integer sum,i;
sum := 0;
for i:=N step -1
    until 1 do
        sum := sum+i;
```
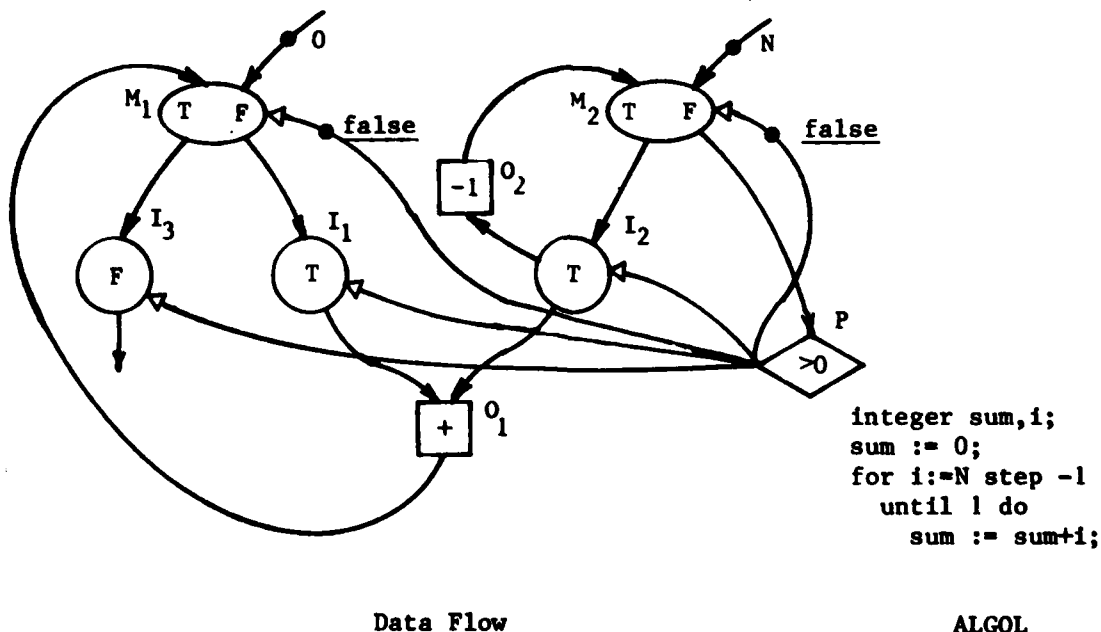
Data Flow   ALGOL

Two Equivalent Programs

Figure 2.1-4

Each of the state transitions occurring in interpreting a program P

from L_B will involve re-distributing the tokens on the input and output

arcs of a single actor in the configuration of P; this is known as

"firing" the actor. The distributions just before and after a firing of

an actor depend on the type of the actor, as depicted in Figure 2.1-5.

The state-transition rule is given in the following two definitions.

Definition 2.1-4 The leftmost of each pair of token distributions shown

in Figure 2.1-5 is the enabled condition for a particular type of actor.

In general, an actor is enabled (to fire) just when all its input arcs

have tokens on them and all its output arcs are empty. The sole

exception is the merge gate. This requires a data token on only one

data input arc; which arc depends on the value of the control token.

$$\triangle$$

Definition 2.1-5 That portion of the state-transition rule of the

standard data-flow interpreter which is pertinent to programs from

L_B is:

   Given a state in a computation sequence, each possible next state

in that sequence is found by:

1. Select one enabled actor d in the configuration of the state.

2. If that actor is one of the types allowed in L_B, then the next

    state is identical except for the input and output arcs of d.

    These are re-configured as in the diagram paired with the

    enabled condition in Figure 2.1-5. The values of the newly-

    created tokens are found as follows, depending on the type of d:
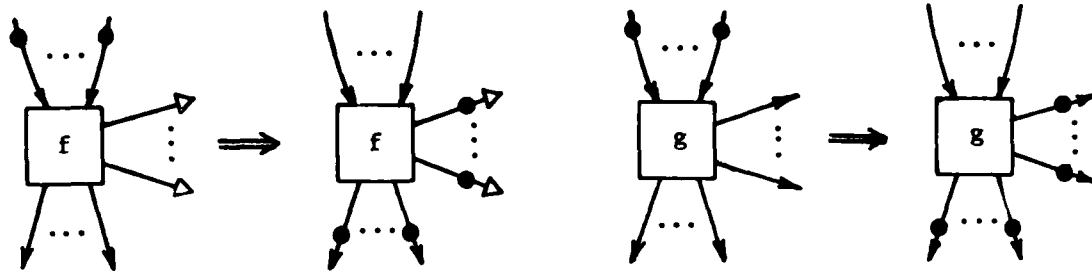
operators
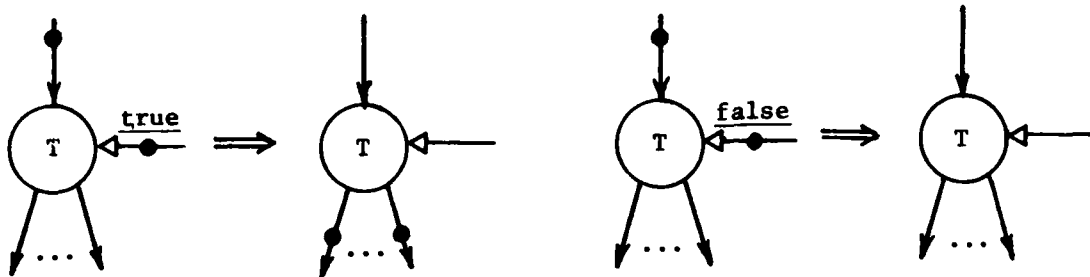
T-gate
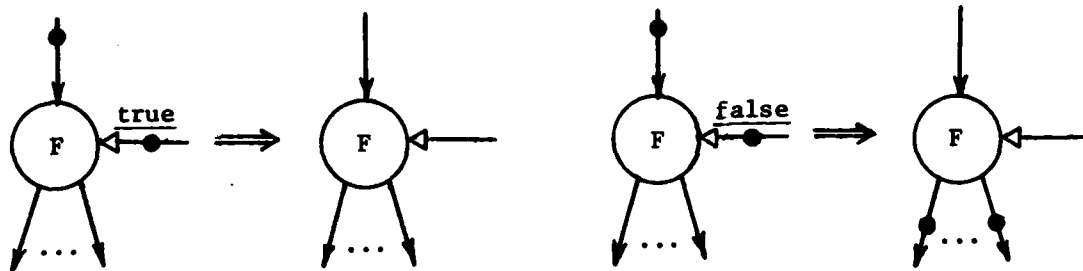
F-gate
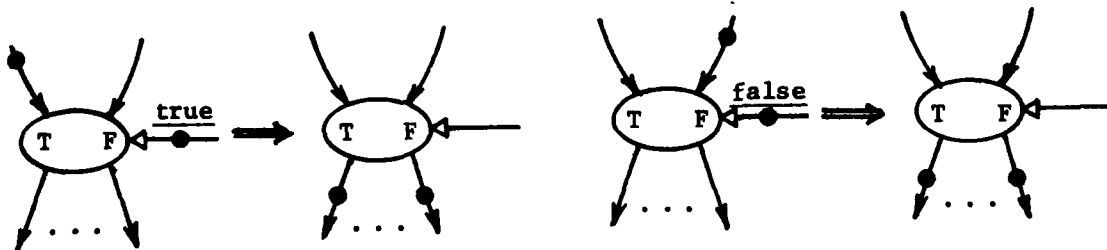
merge gate

Firing Rules for Data-Flow Actors

Figure 2.1-5

and/or   not
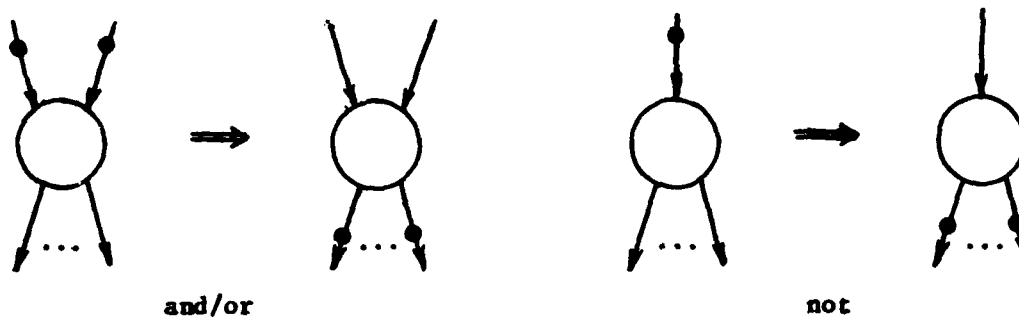
Firing Rules for Data-Flow Actors
Figure 2.1-5 (cont'd)

a. operator - The function associated with d is applied to the

r-tuple of atomic values of the tokens removed from d's

ordered set of r input arcs. The resultant two values are

placed on all of d's number-1 and number-2 output arcs.

b. T-gate and F-gate - The value of the output data token, if any,

is equal to the value of the input data token.

c. merge gate - The value of the output data token is equal to

the value of that input data token which is removed in the

firing. Any token on the other data input arc is undisturbed.

d. Boolean actors - These actors' outputs are defined in the

usual manner.

$$\triangle$$

The gate actors — T-gates, F-gates, and merge gates — are used

together to control flow of data along alternate paths, thereby causing

the performance of alternate computations. Figure 2.1-6 depicts schema-

tically a conditional (if-then-else) construction. This is an m,n program;

the subprograms S_1 and S_2 are m_1,n and m_2,n programs respectively. The

decider D is an m_3,1 program which produces a control output from m_3 data
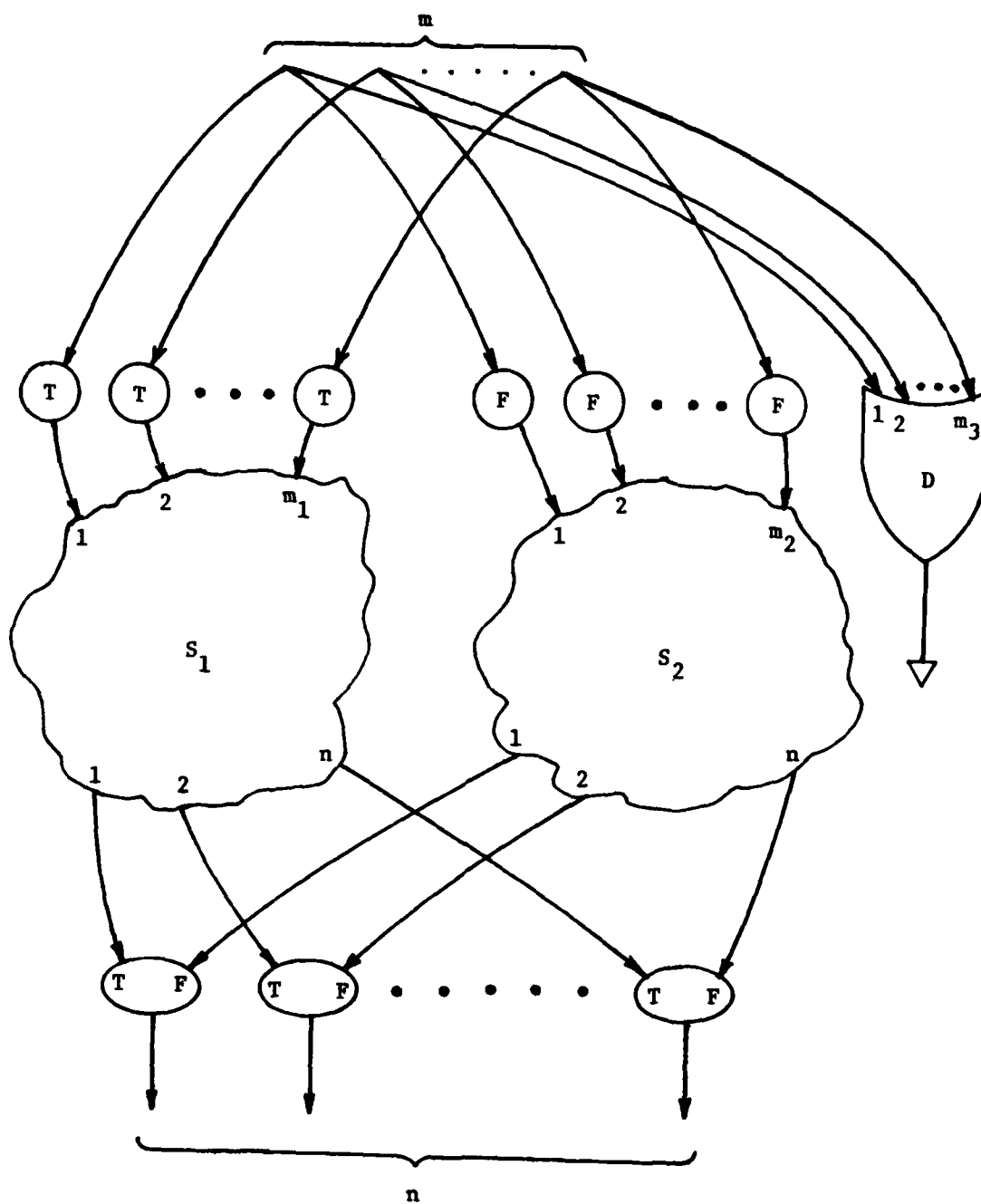
The Conditional Construct

Figure 2.1-6

inputs. Of the m program inputs, a subset of size m_1 are taken through T-gates to become the inputs to S_1, m_2 of them are taken through F-gates to become the inputs to S_2, and m_3 are taken directly to be inputs to D. Each of the m program inputs must be an input to at least one of a T-gate, F-gate, or D. There are n merge gates, each having as inputs one output from each of S_1 and S_2. The connections from the output of D to the control inputs of all the gates have been omitted for clarity. Whenever D outputs a _true_, S_1 gets m_1 inputs from among the m program inputs; the m_2 F-gate inputs simply disappear. The resulting n outputs of S_1 eventually appear on the T inputs of distinct merge gates. Since these gates also have _true_ inputs, the n program outputs are those produced by S_1.

Gates are also used to form the iteration construct, a specific example of which is found in Figure 2.1-4. The interpretation of this program can be explained as follows: In any configuration, just one arc on each of the two directed cycles will have a token associated with it. The value of the token in the left-hand cycle is the value of _sum_; the value in the right-hand cycle is the value of _i_. In the initial configuration, both merge gates M_1 and M_2 have _false_ control inputs, conditioning them to output the values found on their F inputs. These latter values are the program inputs 0 and N respectively.

In the initial configuration, just M_1 and M_2 are enabled to fire. Firing M_1 enables P. On all but the Nth iteration, P outputs a _true_. This enables the T-gates I_1 and I_2 to inject the current values of _sum_ and _i_ into their respective loops; it also disables I_3 from producing a program output, and conditions M_1 and M_2 to receive tokens on their

T inputs. O_2 is enabled as soon as I_2 fires; the firing of O_2 places the next value of $\underline{i}$ on M_2's T input. O_1 is enabled after both I_1 and I_2 have fired; its firing places the sum of the current values of $\underline{sum}$ and $\underline{i}$ on M_1's T input.

Thus each pair of firings of M_1 and M_2 causes the eventual re-enabling of those operators, with new inputs equal respectively to the sum of their last outputs, and to one less than M_2's last output. After the $N+1^{st}$ pair of firings, P's input is no longer greater than zero, so all gates get $\underline{false}$ tokens. The T-gates I_1 and I_2 will then choke off the loops, so that M_1 and M_2 do not get T inputs. I_3 will output the most recent value of $\underline{sum}$, which is the program output. Finally, M_1 and M_2 are re-initialized with $\underline{false}$ inputs. All internal (not program input or output arcs) are now as in the initial configuration, so the program is ready to perform the same computation on the next set of program inputs.

This completes specification of the basic data-flow language L_B and of that portion of the standard interpreter involved in interpreting it. The next section now introduces data-flow languages containing two alternative sets of structure operations.

2.2 Structure Operations

This section defines two languages: the data-flow language with structures as storage, L_{BS}, and the data-flow language with structures as values, L_{BV}. Each of these is an extension of L_B. Computations for both are generated by the single standard data-flow interpreter. The state of this interpreter has two components: 1) the configuration, which has

just been defined, and 2) the <u>heap</u>, which is composed of all data structures
which can be processed by subsequent computation.   The heap is defined
first, followed by the definition of the structure operators in L_{BV} and
L_{BS}, which interact with the heap.

2.2.1 The Heap

The heap takes the form of a directed graph with labels on all
branches and atomic values stored at some nodes (Figure 2.2-1). The
labels on the branches, termed <u>selectors</u>, are drawn from a set Σ of atomic
values (typically, Σ consists of the integers and the character strings.)
No two branches emanating from the same node may be labelled with the same
selector. Formally:
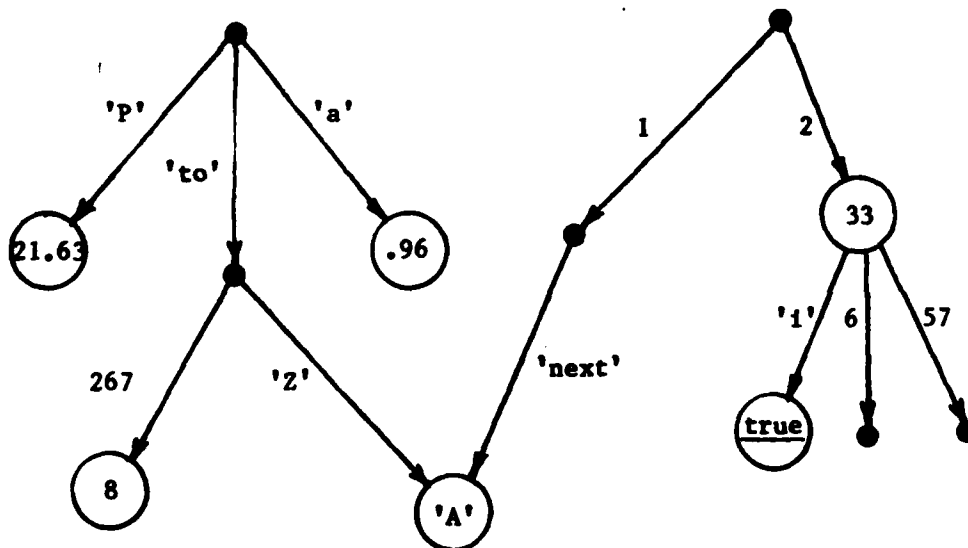
A Heap

Figure 2.2-1

Definition 2.2-1 N is an infinite set of abstract entities called <u>nodes</u>.

V_p is a subset of the atomic value domain V. The elements of V_p are <u>pointers</u>.

The set Σ of <u>selectors</u> is a subset of $V-V_p$, on which has been imposed an arbitrary but fixed total ordering $<$.

The <u>heap</u> component of the state of the standard data-flow interpreter is an ordered triple

$$(N, \Pi, SM)$$

where:

> $N \subset N$ is a finite set of <u>active</u> <u>nodes</u> (the remaining nodes of N
>> are <u>free</u>)
>
> $\Pi: V_p \rightarrow N$
>> is a one-to-one onto mapping from pointers to active nodes.
>
> SM is a function which maps each active node into a <u>content</u>.
>> A content is a set containing:
>>> a) one value from $V-V_p$ or the symbol <u>nil</u>, and
>>>
>>> b) zero or more ordered pairs from $\Sigma \times N$,
>>
>> constrained so that no selector from Σ occurs in more than
>>
>> one pair of the content.

This definition of a heap represents a directed graph by the following correspondences: Atomic value $v \in V$ is the <u>value</u> <u>of</u> active node m iff $v \in SM(m)$. There is a <u>branch</u> from node m to node n labelled with selector s in the heap iff $(s,n) \in SM(m)$.

The notation "dom Π" will be used as an abbreviation for the domain of mapping Π, i.e., the set $\{p \in V_p \mid \exists n \in N: (p,n) \in \Pi\}$.

$\triangle$

Nodes appear only in a heap, where they serve in two capacities: 1) as holders of atomic values, and 2) as endpoints for the branches which indicate relations among structures in the heap. Pointers are atomic values, which appear only as the values of tokens in a configuration. As will be seen, each structure operator in either L_{BV} or L_{BS} has at least one pointer input, and accesses the content of one node in the heap; that node is associated with that pointer by the function Π.

Selectors serve to distinguish among the several branches which relate a node directly to other nodes, and they can be considered to name those relations. The forms of selectors should mimic those actually used in programming systems. These would include integers (used to relate arrays to sub-arrays and to individual elements, for example) and character strings (used to name more general relations.)

The following defines some relationships within a heap:

Definition 2.2-2 For any active node m in a heap:

If there is a branch from m to n, then m is the superior node and n the inferior node of that branch. The set of selectors in all ordered pairs in SM(m) is the set O(m) of selectors off m. The successors of m are just those nodes in ordered pairs in SM(m). For each ordered pair (s,n) in SM(m), n is the s-successor of m.

A path from node m to node n is a sequence of nodes n_1, n_2,...., n_k such that

   i) $n_1 = m$

  ii) $n_k = n$

 iii) for $i = 2,...,k$, n_i is a successor of n_{i-1}.

Node n is <u>reachable</u> <u>from</u> node m iff there exists a path from m to n. The node m, together with all and only those nodes reachable from m, constitute the <u>component</u> <u>rooted</u> <u>at</u> m.

$\triangle$

2.2.2 The Data-Flow Languages with Structures

This section introduces two sets of structure operators. These are defined here as specific actors in a data-flow language. A structure operator is fired just like an atomic operator from the basic language L_B: tokens are removed from all its input arcs and tokens are placed on all its output arcs. However, while the output of an atomic operator is a fixed function of just its inputs, the output of a structure operator may depend on the current heap as well. Furthermore, the firing of certain structure operators will cause changes in the heap.

Adding the two sets of structure operators to L_B results in the languages L_{BV}, the <u>basic</u> <u>data-flow</u> <u>language</u> <u>with</u> <u>structures</u> <u>as</u> <u>values</u>, and L_{BS}, the <u>basic</u> <u>data-flow</u> <u>language</u> <u>with</u> <u>structures</u> <u>as</u> <u>storage</u>:

<u>Definition 2.2-3</u> L_{BV} is a data-flow language in which all actors are restricted to be from one of the four classes of actors in L_B (Definition 2.1-2), or one of the following:

5a. structure operators - Fetch, Const, First, Next, Select, Append, Remove.

L_{BS} is a data-flow language in which all actors are restricted to be from one of the four classes of actors in L_B, or one of the following:

5b. structure operators - Fetch, Assign, First, Next, Select, Copy, Update, Delete.

$\triangle$

The graphical representations of the structure operators are depicted in Figure 2.2-2. Note that the Fetch, First, Next, and Select operations are common to both languages. (The sets of structure operations in L_{BV} and L_{BS} are extensions of those introduced by Dennis in [12] and [11] respectively. His assumed structures in which a node's content could not contain both an atomic value and branches, and he had no operations equivalent to First and Next.)

Computations (state sequences) are generated from programs in both L_{BV} and L_{BS} by the single standard data-flow interpreter. A portion of the state-transition rule has already been given. Completing it requires specifying both the effect of firing each kind of structure operation and the rules for type compatibility: Pointers are a type of atomic value fundamentally different from non-pointers. The only non-trivial actors which can accept pointers as meaningful inputs are the structure operators; it is not possible, e.g., to perform arithmetic on pointers. The few "trivial" actors, including gates and others to be introduced, have some inputs from which they do not attempt to extract any meaning; those inputs, therefore, may be allowed to be of either pointer or non-pointer type. These actors will be known as the pseudo-identity actors:

Definition 2.2-4 A pseudo-identity (pI) actor is any actor which at every firing necessarily outputs tokens with a value equal to that of the token removed from one of its input arcs at that firing. Any input arc of a pI actor whose value could be copied to the actor's output arcs is a transmitted-input arc (i.e., the merge gate is the only pI actor with more than one transmitted-input arc).

$\triangle$
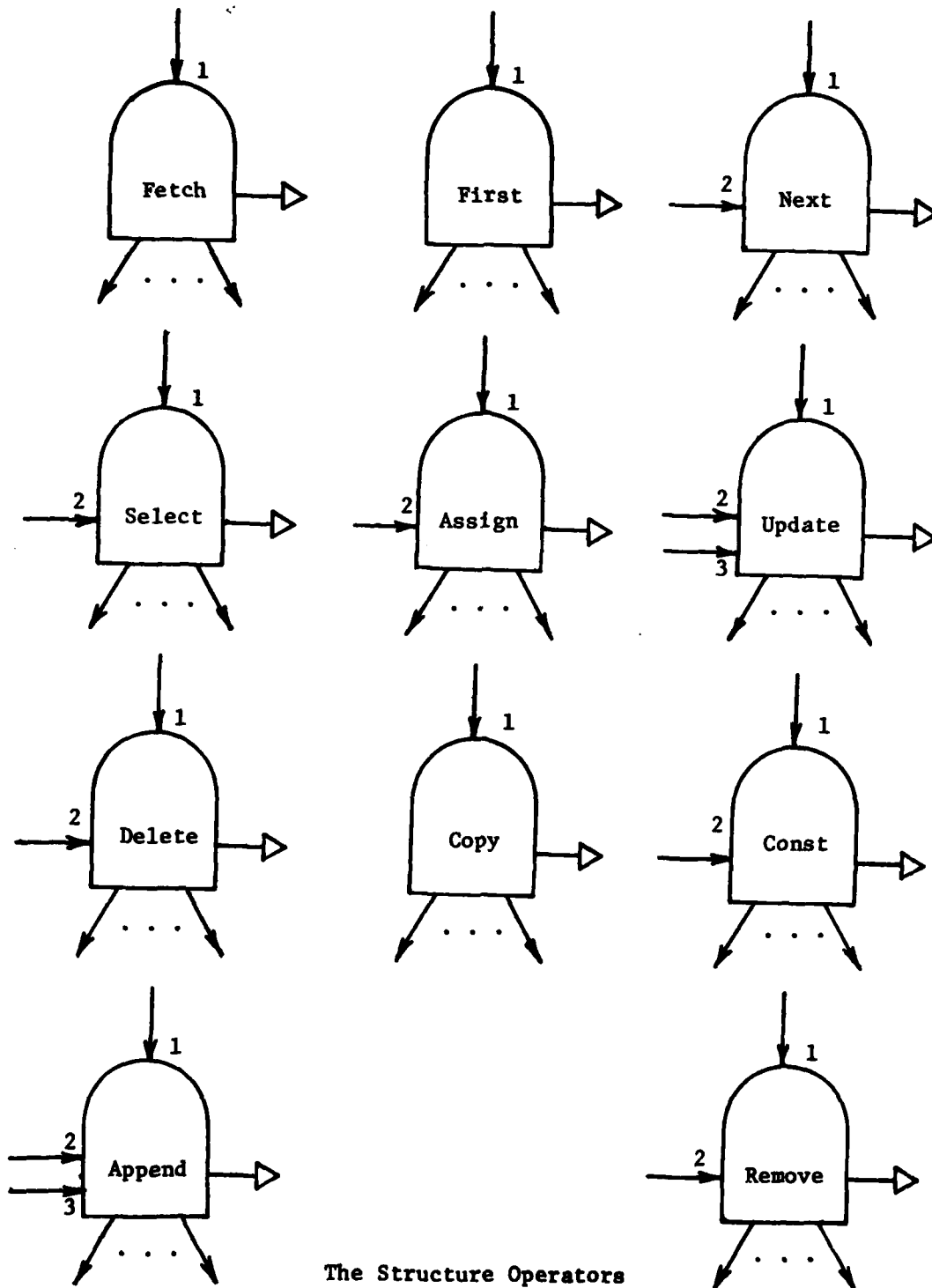
The Structure Operators

Figure 2.2-2

The standard state-transition rule is completed below by codifying the type-compatibility constraints and describing the effect of firing each structure operator (an informal discussion of these operators then follows).

Definition 2.2-5 The <u>state-transition rule</u> of the standard interpreter consists of the portion found in Definition 2.1-5, plus the following:

3. If either

   a. the enabled actor d is not a structure operator or a pI actor, and there is some input arc of d with a token whose value is a pointer, or

   b. d is a structure operator and the values of the tokens on d's input arcs are not as specified in Table 2.2-1 for the type of d, then the next state is a <u>fault</u> state. (The handling of faults is beyond the scope of this thesis.)

4. Otherwise, the next state is related to the current one as follows:

   a. The configuration component is identical except for the input and output arcs of d. The input arcs all have been emptied, and the output arcs all have had tokens placed on them. The value of the tokens placed on the number-2 output arcs is found by evaluating the predicate listed for d in Table 2.2-1. (The Copy operator is unique in that there is no meaningful predicate to associate with it. Hence, both of its groups of output arcs are data arcs, which receive identical output values.) The value of the data output tokens depends on d as follows:

Structure Operator	L_B-[1]	Input Values	Output Values		Heap Alterations	
			Number-1 (Data)	Number-2 (Control)	New[2] Node	New Content
Fetch	S,V	1. $p \in V_p$	v	$v \neq \underline{nil}$	No	SM'(m) = SM(m)
First	S,V	1. $p \in V_p$	s, where $s \in O(m)$ and $\not\exists s' \in O(m): s' < s$	$O(m) \neq \phi$	No	SM'(m) = SM(m)
Next	S,V	1. $p \in V_p$ 2. $s \in \Sigma$	s', where $s' \in O(m)$, $s < s'$, and $\not\exists s'' \in O(m): s < s'' < s'$	$\exists s' \in O(m): s < s'$	No	SM'(m) = SM(m)
Select	S,V	1. $p \in V_p$ 2. $s \in \Sigma$	r, where $(s, \Pi(r)) \in SM(m)$	$s \in O(m)$	No	SM'(m) = SM(m)
Assign	S	1. $p \in V_p$ 2. $v' \in (V - V_p) \cup \{\underline{nil}\}$	0	$v \neq \underline{nil}$	No	SM'(m) = $\{v'\} \cup B$
Update	S	1. $p \in V_p$ 2. $s \in \Sigma$ 3. $r \in V_p$	0	$s \in O(m)$	No	SM'(m) = $\{v\} \cup B^- \cup \{(s, \Pi(r))\}$
Delete	S	1. $p \in V_p$ 2. $s \in \Sigma$	0	$s \in O(m)$	No	SM'(m) = $\{v\} \cup B^-$
Copy	S	1. $p \in V_p$	q, where $q \notin dom\ \Pi$	q	Yes	SM'(n) = SM(m)
Const	V	1. $p \in V_p$ 2. $v' \in (V - V_p) \cup \{\underline{nil}\}$	"	$v \neq \underline{nil}$	Yes	SM'(n) = $\{v'\} \cup B$
Append	V	1. $p \in V_p$ 2. $s \in \Sigma$ 3. $r \in V_p$	"	$s \in O(m)$	Yes	SM'(n) = $\{v\} \cup B^- \cup \{(s, \Pi(r))\}$
Remove	V	1. $p \in V_p$ 2. $s \in \Sigma$	"	$s \in O(m)$	Yes	SM'(n) = $\{v\} \cup B^-$

Legend -

   (N, Π, SM) is the current heap

   (N', Π', SM') is the new heap

   $m = \Pi(p)$

   v is the unique value from $V - V_p \cup \{\underline{nil}\}$

     which is in SM(m)

   $O(m) = \{s \mid \exists n: (s,n) \in SM(m)\}$

   $B = \{(s',n) \mid (s',n) \in SM(m)\}$

   $B^- = \{(s',n) \mid (s',n) \in SM(m) \wedge s' \neq s\}$

Notes -

  1. S $\Rightarrow$ operator is in L_{BS}

      V $\Rightarrow$ operator is in L_{BV}

  2. No $\Rightarrow$ N' = N, Π' = Π, $(\forall m' \neq m)(SM'(m') = SM(m'))$

      Yes $\Rightarrow$ N' = N $\cup \{n\}$ where $n \in N'$-N,

           $\forall r \in dom\ \Pi$, $\Pi'(r) = \Pi(r)$, $\Pi'(q) = n$

           $\forall m' \in N$, SM'(m') = SM(m')

Specifications of the Structure Operations

Table 2.2-1

Fetch, First, Next, Select - If the control output value is
false, then the data output tokens have the value undef. Other-
wise, they depend on the current heap as indicated in the Table.

Assign, Update, Delete - The data output is identically zero.
This data output token can be used for synchronization, as will
be seen in Chapter 3.

Copy, Const, Append, Remove - The data outputs are equal to q, an
arbitrary pointer not in the domain of the current Π.

b. The new heap component depends on d as indicated in Table 2.2-1
under Heap Alterations. These dependencies can be categorized:

Fetch, First, Next, Select - The new heap is identical to the
old one.

Assign, Update, Delete - The only difference is a modification
in the content of the node $m = \Pi(p)$, which was active in the
current heap.

Copy, Const, Append, Remove - An arbitrary free node n is
activated: The set N of active nodes in the heap is augmented
by n, the domain of the function Π is augmented by the pointer
q which is the data output, and $\Pi(q) = n$. The content of n is
a close derivative of m's content.

$\triangle$

Below is an informal discussion of the usefulness of this particular
selection of structure operators; following that is a formal characteri-
zation of the state of the interpreter during a computation sequence.

The decomposition operators — Fetch, First, Next, and Select — are
common to both L_{BV} and L_{BS}. Fetch, given a pointer p, outputs the value

of the node $m = \Pi(p)$. The First and Next operators allow enumeration of
the set $O(m)$ of selectors labelling the branches emanating from m. These
operators sort $O(m)$ according to the assumed total ordering $<$ on the entire
set Σ of selectors: First outputs the least selector in $O(m)$, and Next
inputs one selector and outputs the next-greater selector in $O(m)$.
Enumeration is accomplished by applying First once, and then Next repeti-
tively, until a <u>false</u> control output obtains.

The Select operator inputs a pointer p and a selector s and outputs
a pointer to the s-successor of $m = \Pi(p)$ (if one exists). The set of
successors of m may be discovered by applying Select to each selector in
the enumeration of $O(m)$. Recursive application of this procedure leads to
the discovery of all nodes reachable from m, and of all branches between
any two such nodes. Thus complete decomposition of any given component is
straightforward in both L_{BV} and L_{BS}.

The remaining operators in each language are its <u>construction</u>
<u>operators</u>. These are capable of constructing in the heap any arbitrary
component. The operators in L_{BV} have been chosen in the expectation that
most components constructed will be very similar to existing ones. There-
fore, each operator activates a new node whose content differs minimally
from an existing node's content: Const activates a node whose content has
a given value, but is otherwise identical to the content of a given node.
Append activates a node whose content is distinguished from a given node's
only by the presence of a given ordered pair (and the consequent absence of
any other pair with the same selector). Remové is provided to activate
a node whose content is distinguished by the absence of any pair with a
given selector.

The structure operations chosen for L_{BV} have intentionally been kept simple, compared with those offered in [1] and [31], which are oriented more toward efficiency in both programming and implementation. The advantages of simple operations are that (1) they are formally more tractable, and (2) they constitute a more general basis for composing various sets of complex operations.

The structure operations in L_{BS} also exhibit these advantages to substantially the same degree. The decomposition operations are identical to those in L_{BV}. For every construction operator in L_{BV}, there is a two-operator combination in L_{BS} which has the same effect. Figure 2.2-3 illustrates this for the Const operator: The Copy operator activates a
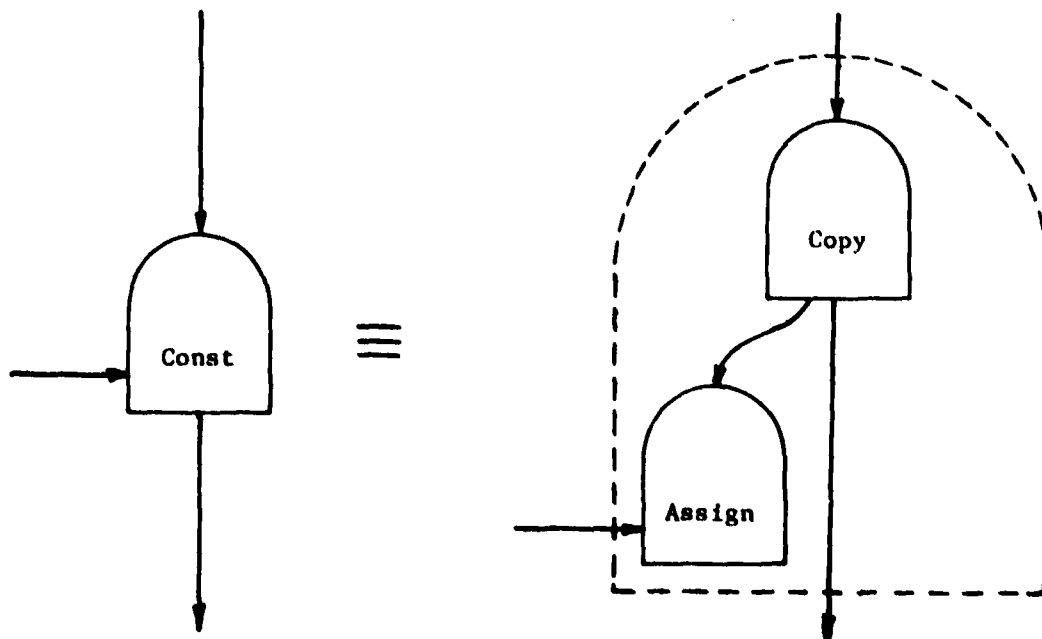
Equivalence of L_{BS} to L_{BV}

Figure 2.2-3

new node n with a content identical to that of its input node m. The

Assign operator modifies the content of its input node n, giving it a new

value v'. Thus the heap is altered by this L_{BS} combination in exactly

the same way as by the single L_{BV} operator Const.

The fundamental difference between L_{BV} and L_{BS} is in this:

Is the content of a node altered before or after the pointer

to that node appears as the value of any tokens?

In L_{BV}, the node is always altered before; in L_{BS}, it is always altered

after. This means that in an implementation of L_{BS}, the physical process

of constructing a new component can be partially overlapped in time with

the process of decomposing that same component. This phenomenon, which

may be called "structure concurrency", cannot occur in L_{BV}. As illustrated

by the example programs in the next section, structure concurrency has

two vital consequences:

1. An L_{BS} program has the potential for more concurrency, hence a

    shorter minimum execution time, than an equivalent L_{BV} program.

2. The L_{BS} program potentially produces the wrong result.

This section concludes with a study of properties of the interpreter state

which are preserved by the state-transition rule.

2.2.3 Formal Semantics

An interpreter generates a set of computations from a program P and

an input to P in the following manner: P plus its input establish an

initial state for the interpreter according to some convention. Each

computation is a sequence of interpreter states generated from the initial

state by repeated applications of a state-transition rule. The state-

transition rule for the standard interpreter has already been fully specified. A convention by which a program together with an input to it establish an initial state is provided below. This is followed by a demonstration that the state-transition rule, particularly the definitions of the structure operations, is consistent in the following sense: In each state in a computation sequence, the second component of the state truly is a heap, and each pointer in the configuration points to a node in the heap component.

The initial state in any computation sequence in the standard interpreter will satisfy the following specification:

Definition 2.2-6 An initial state of the standard data-flow interpreter for any program P is a pair (Γ, U), where

$\quad\quad\quad \Gamma$ is a configuration of P, and

$\quad\quad\quad U = (N, \Pi, SM)$ is a heap,

satisfying

1. there are in Γ data tokens on all program input arcs of P and on no other data arcs, and

2. every pointer which is the value of one of these tokens is in the domain of Π.

An initial state for P establishes values for all of the program inputs to P by the following correspondence: If an input arc holds a token with a non-pointer value, then the corresponding program input is that value. If an arc holds a token with a pointer value p, then the corresponding program input is the data structure which is the entire component rooted at $\Pi(p)$.

Theorem 2.2-1 Let S_0 be any initial standard interpreter state for an L_{BS} program, and let $S = (\Gamma,(N,\Pi,SM))$ be any final state in a sequence derived from S_0 by repeated applications of the state-transition rule. Then:

A: For each $n \in N$, and for any $s \in \Sigma$, $(s,m) \in SM(n) \Rightarrow m \in N$.

B: There is a token with value $p \in V_p$ on an arc in Γ only if $p \in dom \, \Pi$.

C: Π is one-to-one onto N.

Proof: By induction on the length of the state sequence.

Basis: The length of the sequence is one; i.e., $S = S_0$, the initial state.

(1) (N,Π,SM) is a heap and B   Def. 2.2-6

(2) A and C   (1)+Def. 2.2-1

Induction step: Assume that A, B, and C are true for the final state in any sequence of length $n > 0$, and consider a sequence of length $n+1$. Let the final state in that latter sequence be S'.

(3) S' is derived by applying the state-transition rule once to a
      state S, which is the final state in a sequence of length n

Let $S = (\Gamma,(N,\Pi,SM))$ and $S' = (\Gamma',(N',\Pi',SM'))$. Let d be the enabled actor chosen to fire in the transition from S to S'. There are four cases to consider, depending on the type of actor d is.

Case I: d is not a Select, Update, or Copy.

(4) $\Pi' = \Pi$ and $N' = N$, so Π' is onto N' (3)+Def. 2.2-5+ind. hyp. C

(5) For any $n \in N'$, let (s,m) be any ordered pair in $SM'(n)$. Then $n \in N$
      and $(s,m) \in SM(n)$ (4)+Def. 2.2-5

(6) $m \in N$ (5)+(3)+ind. hyp. A

(7) $m \in N'$ (6)+(4)

(8) There is a token with pointer value p on an arc in $\Gamma' \Rightarrow$ there is

a token with value p on an arc in Γ Def. 2.2-5

(9) $\Rightarrow$ p$\in$dom Π $\Rightarrow$ p$\in$dom Π' (3)+(4)+ind. hyp. B

Case II: d is a Select

(10) Π' = Π, N' = N, and SM' = SM   Def. 2.2-5

(11) For any n$\in$N', (s,m) is any pair in SM'(n) $\Rightarrow$ n$\in$N and (s,m)$\in$SM(n) (10)

(12) $\Rightarrow$ m$\in$N $\Rightarrow$ m$\in$N'   (10)+(3)+ind. hyp. A

(13) There is a token with pointer value p on an arc in Γ' $\Rightarrow$ there is

     a token with value p in Γ or that token was placed on a data

     output arc of d in the transition   Def. 2.2-5

(14) There is a token with value p in Γ $\Rightarrow$ p$\in$dom Π' (10)+(3)+ind. hyp. B

(15) Let q and s be the values of the tokens removed from d's pointer

     and selector input arcs. Then a token of value p was placed on

     d's output arcs $\Rightarrow$ there is a pair (s,m) in SM(Π(q))   Def. 2.2-5

(16) $\Rightarrow$ m$\in$N   (3)+ind. hyp. A

(17) $\Rightarrow$ p$\in$dom Π $\Rightarrow$ p$\in$dom Π' (10)+(3)+ind. hyp. C

(18) Π' is onto N'   (10)+(3)+ind. hyp. C

Case III: d is an Update

(19) Π' = Π and N' = N, so Π' is onto N'   (3)+Def. 2.2-5+ind. hyp. C

(20) For any n$\in$N', (s,m)$\in$SM'(n) $\Rightarrow$ (s,m)$\in$SM(n) or m = Π(r) where pointer

     r is the value of a token on an input arc of d in Γ Def. 2.2-5

(21) (s,m)$\in$SM(n) $\Rightarrow$ m$\in$N $\Rightarrow$ m$\in$N'   (19)+(3)+ind. hyp. A

(22) r is the value of a token on an arc in Γ $\Rightarrow$ r$\in$dom Π (3)+ind. hyp. '

(23) $\Rightarrow$ m = Π(r) is in N = N'   (20)+(19)+(3)+ind. hyp. C

(24) There is a token with pointer value p on an arc in Γ' $\Rightarrow$ there is

     a token with value p on an arc in Γ Def. 2.2-5

(25) $\Rightarrow$ p$\in$dom Π $\Rightarrow$ p$\in$dom Π' (19)+(3)+ind. hyp. B

Case IV: d is a Copy

(26) $\Pi' = \Pi \cup \{(p,n)\}$ and $N' = N \cup \{n\}$, where $(p,n) \notin \Pi$ and $n \notin N$, and p is

      placed on an output arc of d in the transition   Def. 2.2-5

(27) Π' is onto N' (26)+(3)+ind. hyp. C

(28) For all $n' \neq n$ in N', $(s,m) \in SM'(n') \Rightarrow (s,m) \in SM(n')$ (26)+Def. 2.2-5

(29) $\Rightarrow m \in N \Rightarrow m \in N'$ (26)+(3)+ind. hyp. A

(30) Let r be the pointer value of the token removed from d's input

      arc in the transition. Then $SM'(n) = SM(\Pi(r))$ Def. 2.2-5

(31) $(s,m) \in SM'(n) \Rightarrow (s,m) \in SM(\Pi(r)) \Rightarrow m \in n \Rightarrow m \in N'$ (30)+(26)+(3)+ind. hyp. A

(32) There is a token with value q on an arc in $\Gamma' \Rightarrow$ there is a token

      with value q on an arc in Γ or $q = p$ (26)+Def. 2.2-5

(33) There is a token with pointer value q on an arc in $\Gamma \Rightarrow q \in dom \, \Pi \Rightarrow$

      $q \in dom \, \Pi'$ and $q \neq p$ (26)+(3)+ind. hyp. B

(34) $q = p \Rightarrow q \in dom \, \Pi'$ (26)

(35) B for Γ' (32)+(33)+(34)

$$\triangle$$

2.3 Computations Over Structures

This section first presents two simple data-flow programs with

structures: AlterV, written in L_{BV}, and AlterS, written in L_{BS}. These

will be used to illustrate how the standard data-flow interpreter gives

meaning to programs with structures. Additionally, AlterS is the

simplest program in which the phenomenon of structure concurrency is

observed to cause incorrect results. The other alleged consequence of

structure concurrency, reduced execution time, is not evident in programs

as small as these; therefore, this effect is studied in a pair of larger

programs.

2.3.1 A Simple L_{BV} Program

Figure 2.3-1 shows the L_{BV} program AlterV. This program has two distinct inputs. The X input must be a pointer to a node in the heap, and the Y input must be a non-pointer value. The only non-structure operator in this program is the <u>constant generator</u> G. This operator ignores the value of its one input, which here is a program input. Its output arcs are the selector input arcs of S_1, S_2, and A; these have not been connected, to avoid confusion. Each time G fires, it places tokens with the constant selector value 'next' on all these arcs.

The intent of AlterV can be understood informally as follows: The first part, consisting of operators S_1, C, and A, constructs a component identical to that pointed to by the X input, except for this: The 'next'-successor of the root node has a value equal to the Y input. The program output Q is a pointer to the root of this new component. The second part of the program, consisting of S_2 and F, fetches the value of the 'next'-successor of the root node of the newly-created component. Therefore, the program output R should equal the program input Y.

Figure 2.3-2 depicts an initial state S for the program AlterV. The configuration is shown on the left, the heap on the right. (The labels m_1 and m_2 on the nodes in the heap are for reference purposes only.) The program input Y is 3. The program input X is a pointer p to node m_1 in the heap; this is indicated by the arrows to m_1 from the tokens on the program input arcs.

If the interpreter is started in state S, successive applications of the state-transition rule will take the interpreter through a sequence of
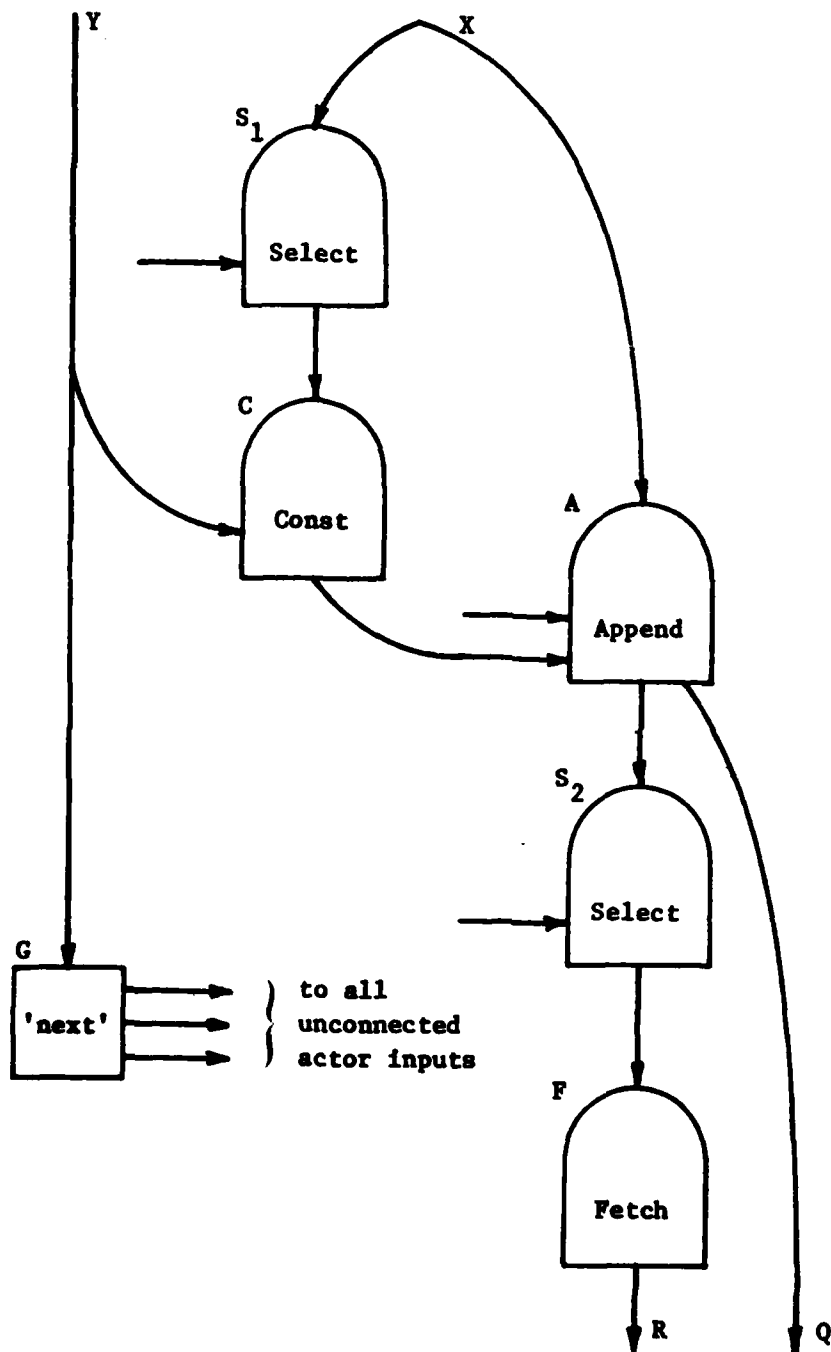
The L_{BV} Program AlterV

Figure 2.3-1
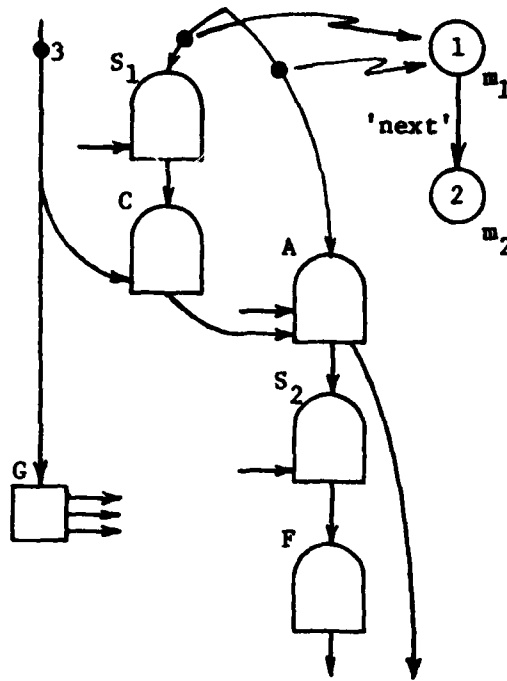
An Initial State S for AlterV

Figure 2.3-2

states. Figure 2.3-3 shows several states in this sequence. The only actor enabled in the initial state is the constant generator G. Firing this places the selector 'next' on all selector input arcs, resulting in a state in which only Select S_1 is enabled. Firing S_1 results in the state shown in Figure 2.3-3(a). S_1's inputs were a pointer to m_1 and the selector 'next'; its output is a pointer to m_2, the 'next'-successor of m_1. Part (b) shows the result of firing Const operator C. A new node n_2 is activated with a value of 3. The Append A is the only operator enabled in this state; the state after it fires is in part (c). A second
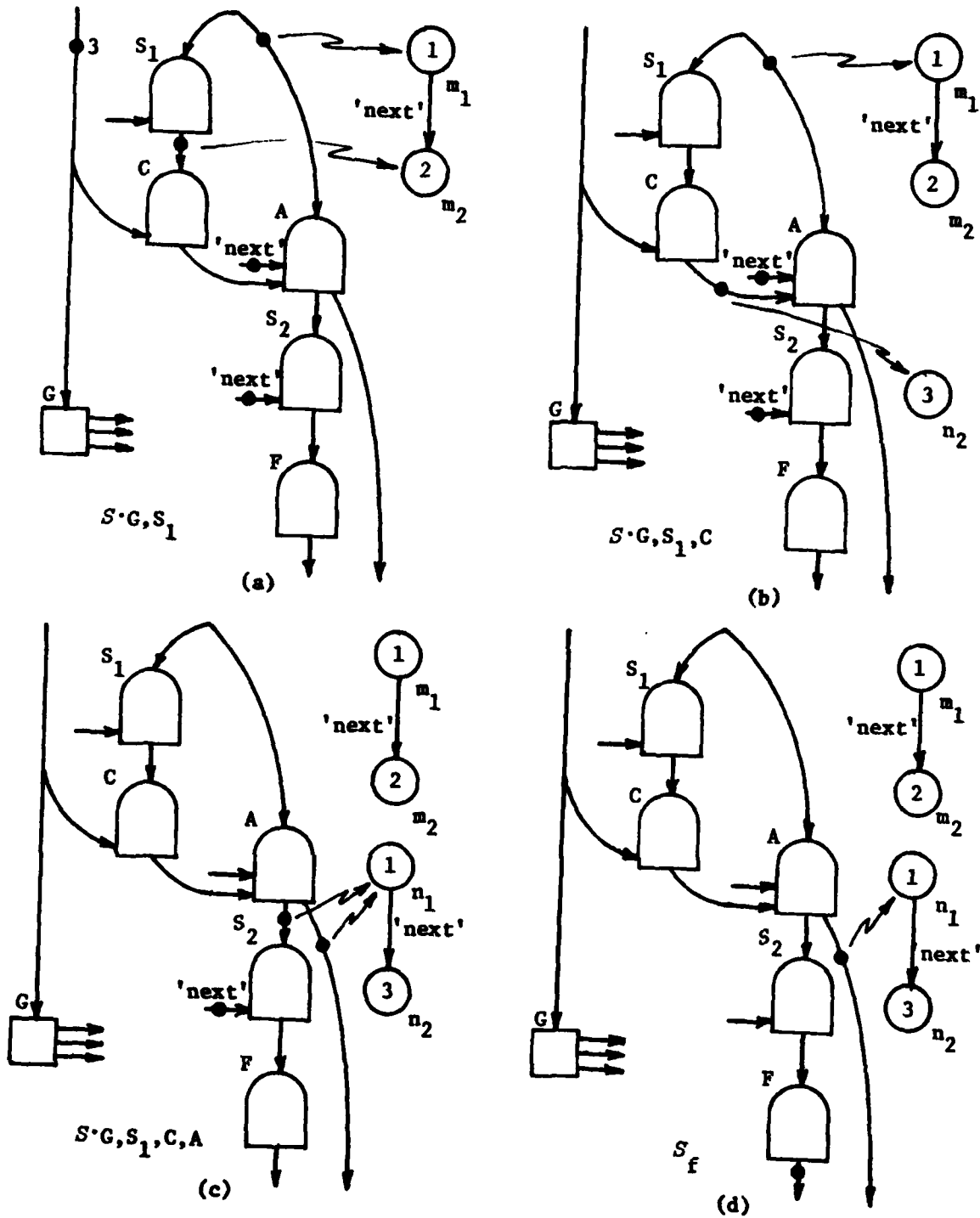
A State Sequence for AlterV

Figure 2.3-3

new node n_1 has been activated. This node has the same content as m_1, except that its 'next'-successor is n_2 instead of m_2. I.e., the node n_1 is the root of a new component which differs from the program input only in the value of the 'next'-successor of its root. A pointer to this new component is now on the program output arc Q. Firing the remaining two operators results in the final state S_f (Figure 2.3-3(d)). The value 3 has been fetched from n_2, and a token with that value appears on the program output arc R.

This example illustrates the formal derivation of one possible outcome for the given input. The final state in Figure 2.3-3(d) establishes values for the program outputs in a manner analogous to the establishment of program inputs by an initial state. A final state can be found only by using the state-transition rule to generate a sequence of states starting in the initial state. Determining all possible outcomes for a given input is ultimately a matter of generating all possible state sequences starting in all possible initial states which establish that input.

Distinguishing one of these state sequences from another by comparing graphical representations, like those in Figure 2.3-3, is unworkable. A convenient abbreviation for a state sequence is a _firing sequence_. This is basically the sequence of the labels of the actors fired at each state transition. An entire state sequence can be uniquely re-constructed from its initial state and its firing sequence, as in the following:

Definition 2.3-1 Let S be any state for a data-flow program P. Let d be the label of any actor in P. Then a _firing_ φ _of the actor_

<u>labelled</u> d (or a <u>firing</u> <u>of</u> d, for short) is defined by

$$\varphi = \begin{cases} d & \text{if the actor is not a Copy, Const, Append, or Remove} \\ (d,(p,n)) & \text{otherwise, where p is any pointer and n is any node} \end{cases}$$

A <u>firing</u> <u>sequence</u> <u>starting</u> <u>in</u> S, and the <u>state</u> <u>after</u> <u>firing</u> <u>sequence</u> Ω, $S \cdot \Omega$, are jointly defined by the following recursive rules:

(1) λ, the empty sequence, is a firing sequence starting in S.

$S \cdot \lambda = S$.

(2) Let $\Omega = \varphi_1, \varphi_2, \ldots, \varphi_{n-1}$ be a firing sequence starting in S, and let

d be the label of any actor enabled in $S \cdot \Omega$. Then

$\Omega\varphi_n = \varphi_1, \varphi_2, \ldots, \varphi_{n-1}, \varphi_n$, where φ_n is a firing of d, is a firing

sequence starting in S.

$S \cdot \Omega\varphi_n$ is the state obtained by applying the state-transition rule

to $S \cdot \Omega$ with d as the enabled actor selected to fire. If d is a Copy,

Const, Append, or Remove, then it is the ordered pair (p,n) which

is added to Π.

(3) All firing sequences starting in S are defined by (1) and (2) above.

Any firing sequence Ω is <u>halted</u> iff no actor is enabled in $S \cdot \Omega$.

$\triangle$

The only freedom of choice in the application of the state-transition

rule is in the selection of:

1. which enabled actor is fired,

and if a Copy, Const, Append, or Remove is chosen,

2. what pointer-node pair is added to the function Π.

Each possible choice can be expressed as a unique firing. Any state plus

a firing of an actor enabled in that state determines a unique next state.

Thus a sequence of firings starting in an initial state uniquely

determines a sequence of states, as in the above definition.

An initial state for a program P establishes some set of program input values for P. The data-flow interpreter associates with each such initial state a set of possible state/firing sequences. The final state entailed by each such sequence establishes a set of program output values for P, at least if P is well-behaved [12]:

Definition 2.3-2 A data-flow program program P is <u>well-behaved</u> iff the following is true of every initial state S for P: Let Ω be any halted firing sequence starting in S and let (Γ, U) be the state $S \cdot \Omega$. Then in Γ:

1. Every program input arc of P has no tokens on it.

2. Every program output arc of P has a token on it.

3. Every other arc is configured exactly as in S.

$\triangle$

Thus the interpreter associates one or more sets of output values with each possible set of input values for a program. AlterV is a program for which each set of inputs has exactly one set of outputs associated with it. For the program AlterS, presented next, a given set of inputs may have many different sets of outputs associated with it.

2.3.2 The L_{BS} Program AlterS

AlterS (Figure 2.3-4) illustrates the hazards of structure concurrency. It is derived from AlterV by performing the substitution shown in Figure 2.2-3 for the Const, and a similar one for the Append. It is argued at the end of Section 2.2 that the substituted combinations change the heap in the same way as the L_{BV} operators which they replace. It is
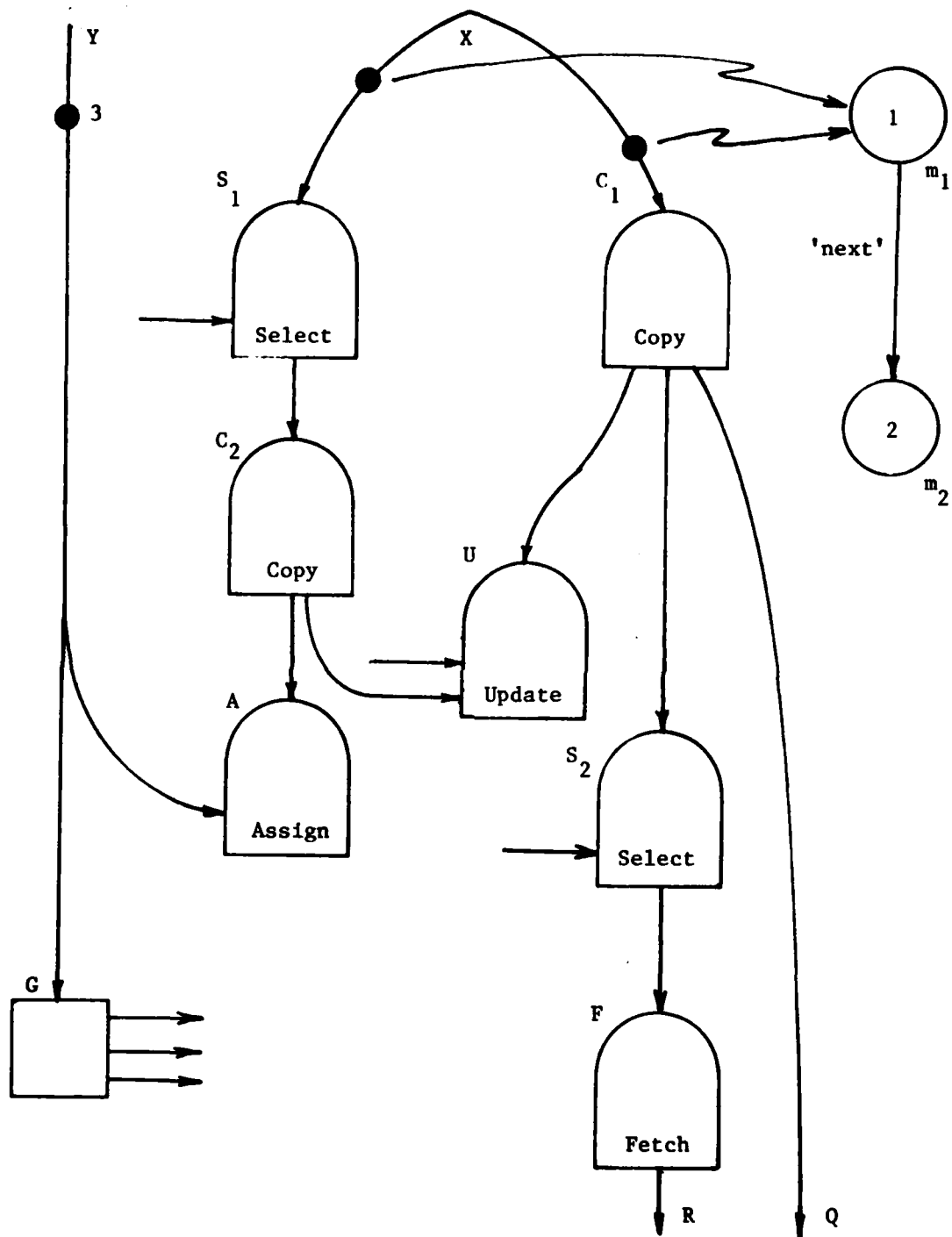
The L$_{BS}$ Program AlterS

Figure 2.3-4

therefore reasonable to expect that AlterS and AlterV will always yield "equal" results when applied to the same inputs. ("Equality" is used here in an intuitive sense; a formal definition is given in Section 2.4.) Unfortunately, that is not the case. Different firing sequences starting in the same initial state for AlterS may yield unequal final states, as is demonstrated next.

The initial state $\mathcal{S}$ is that shown in Figure 2.3-4. The program inputs are equal to the inputs of AlterV in the initial state of Figure 2.3-2. Consider first the firing sequence $\mathcal{S}_1 = G, S_1, C_1, C_2, U, A, S_2, F$. Figure 2.3-5 shows the interpreter state after selected prefixes of $\mathcal{S}_1'$. In $\mathcal{S} \cdot G, S_1, C_1$ (part (a) of the Figure), the output arc of S_1 has a pointer to node m_2. The output arcs of C_1 have pointers to n_1, which is a copy of m_1 (i.e., is a newly-activated node having the same content as m_1.) In $\mathcal{S} \cdot G, S_1, C_1, C_2, U$ (part (b)), a copy n_2 has been made of node m_2, and the 'next'-successor of n_1 has been changed by the Update to be n_2.

In $\mathcal{S} \cdot G, S_1, C_1, C_2, U, A$ (part (c)), the value of n_2 has been changed by the Assign to be 3. The program output arc Q has a pointer to n_1, which is the root of a component differing from the program input X only in the value of the root's 'next'-successor. Firing S_2 and F fetches the value 3 from n_2, resulting in the final state $\mathcal{S} \cdot \mathcal{S}_1$, shown in part (d).

It is apparent that the outputs of AlterS in $\mathcal{S} \cdot \mathcal{S}_1'$ are equal to the unique outputs produced by AlterV for the same inputs. Consider however the firing sequence $\mathcal{S}_2 = G, S_1, C_1, C_2, S_2, U, A, F$. In the state $\mathcal{S} \cdot G, S_1, C_1, C_2$ (Figure 2.3-6), S_2 is enabled with a pointer to n_1 as input. If S_2 is fired in this state, a pointer to node m_2 will be placed on F's input arc.

[†]All selector inputs and branch labels are 'next'.

(a) $S \cdot G, S_1, C_1$

(b) $S \cdot G, S_1, C_1, C_2, U$

(c) $S \cdot G, S_1, C_1, C_2, U, A$

(d) $S \cdot \Omega_1$

A State Sequence for AlterS

Figure 2.3-5

A State in an Alternative Sequence for AlterS

Figure 2.3-6

Then when F eventually fires, a token with value 2 will be placed on the program output arc R. So the R output arc has a token of value 2 in $S \cdot \Omega_2$ and a token of value 3 in $S \cdot \Omega_1$. There are two firing sequences, starting in the same initial state for AlterS, which lead to final states with unequal program outputs. Any program for which this may be true is non-functional (this property is formally defined in Section 2.4). An L_{BS} program may be non-functional, but all L_{BV} programs are functional.

$S \cdot \Omega_2$ differs from $S \cdot \Omega_1$ because of the concurrent construction and decomposition of the new component rooted at n_1. In $S \cdot G, S_1, C_1, C_2$ (Figure 2.3-6), n_1 has been activated and there are pointers to it available. During subsequent computation, a new component rooted at

n_1 will be constructed. There is in this state a "race" between the construction operator combination U-A and the decomposition combination S_2-F. If the decomposition combination loses this race, then it decomposes the new component, as intended. If it wins, then it decomposes whatever component was originally rooted at n_1, which happens to result in outputting 2, the value of m_2.

Thus structure concurrency may induce non-functional program behavior (under conditions made clear in Section 3.1.1). A potentially-compensating benefit of this concurrency is demonstrated in the next sub-section.

2.3.3 Analysis of Execution Time

Figure 2.3-7 shows two programs: AlterV2, written in L_{BV}, and AlterS2, written in L_{BS}. These programs are similar to AlterV and AlterS. The only difference is in the level at which the output component differs from the input: Each of these programs first constructs a component identical to its X input except for the value of the 'next'-successor of the 'next'-successor of the root. It then fetches this value from the newly-constructed component. (The constant-selector generator has been omitted for simplicity.)

The purpose here is to estimate the relative total elapsed times required to execute AlterV2 and AlterS2. The analysis is based on the following simplifying assumptions:

Assumption 2.3-1 The time required to execute an operation is one of three constant durations, depending on the type of the operation:

a. S is the time required for the Copy, Const, and Append operations. Each of these entails the following sub-operations:
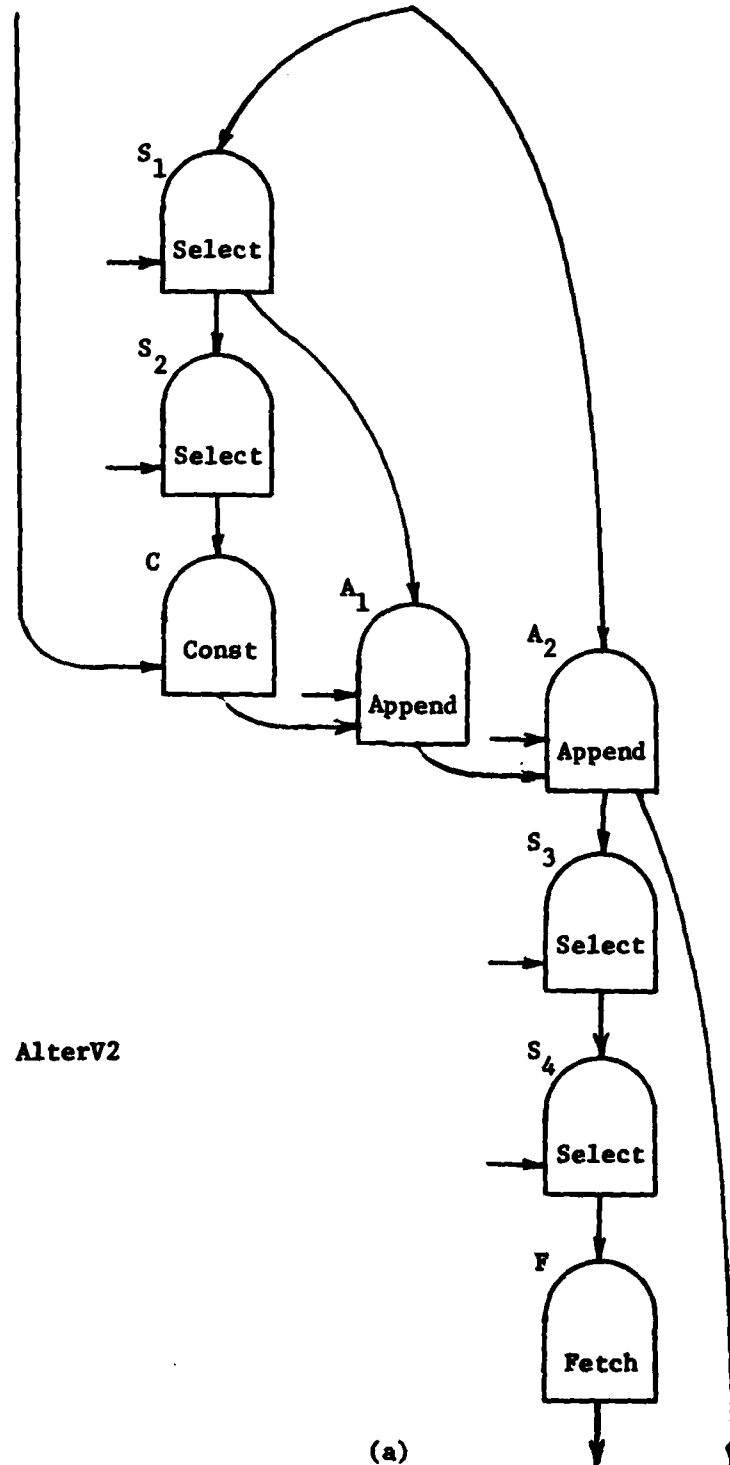
(a)

The Programs AlterV2 and AlterS2
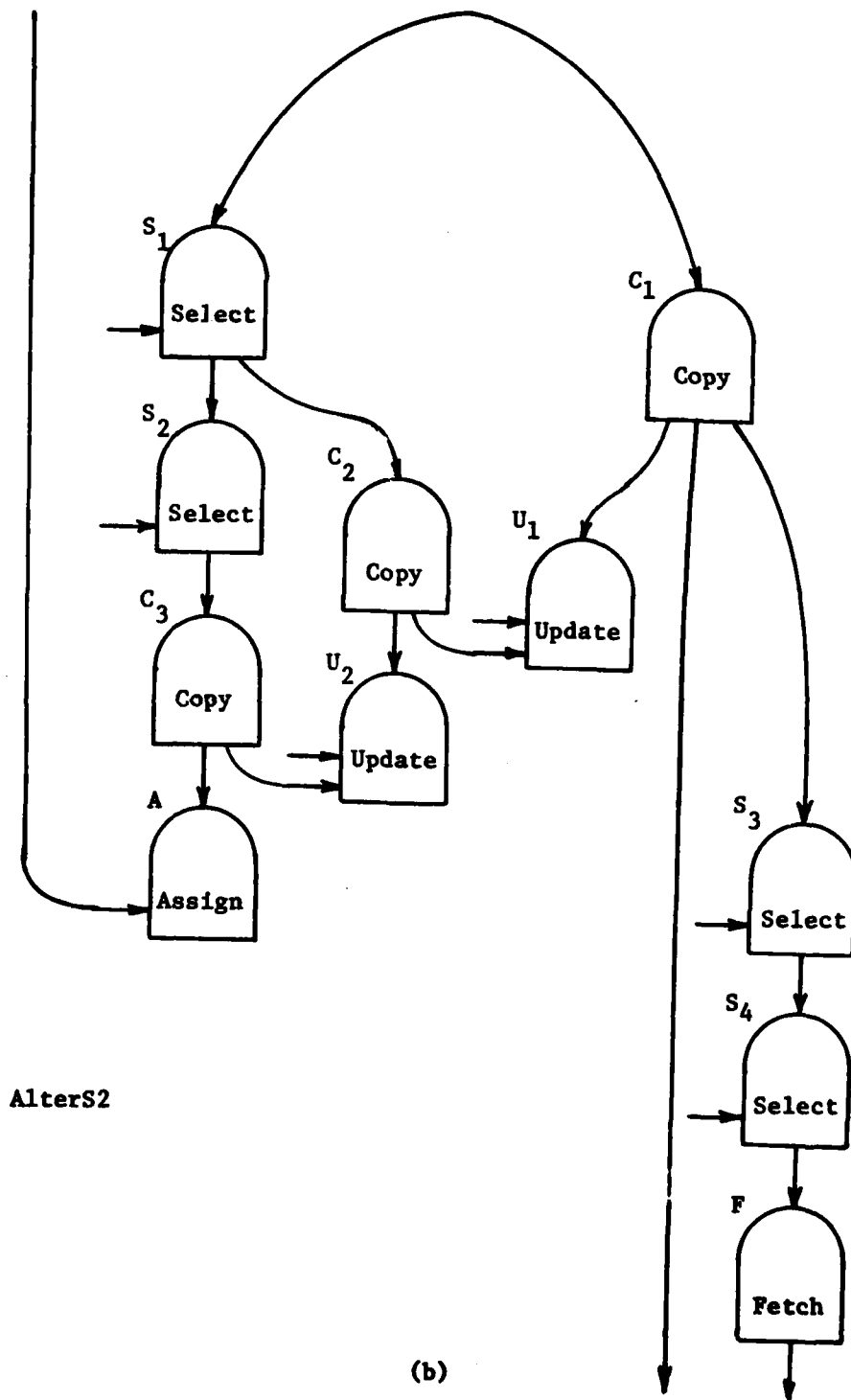
Figure 2.3-7

AlterS2

(b)

The Programs AlterV2 and AlterS2

Figure 2.3-7 (cont'd)

    i. Find where the content of the input node is stored in the
Structure Memory (SM).

    ii. Find an unused pointer value and an empty location in the SM
for a new node's content.

    iii. Read the value and every ordered pair in the old content,
copying it (with possibly one change) into the empty location.

b. P is the time required to execute a Select or Update. Each of these
involves the following steps:

    i. Find the content of the input node.

    ii. Search through the ordered pairs in that content until one
with the given selector is found.

    iii. Either return this (Select) or overwrite it (Update).

c. V is the time required for the Fetch and Assign operations. These
entail the following sub-operations:

    i. Find the content of the input node.

    ii. Read the value in that content, and return it (Fetch) or
overwrite it (Assign).

It is clear that $P \leq S$ and $V \leq S$, and it is likely that $V < P$.

$\triangle$

Assumption 2.3-2 Each operator's execution starts as soon as all of the
other executions which must precede it have finished. This includes those
which supply its inputs, as well as those which must precede it for cor-
rectness; i.e., for AlterS2 to produce the same result as AlterV2, U_1's
execution must finish before S_3's starts. Thus, the total execution time
for a program equals the maximum sum of execution times for all sequences
of operators of the following form: For each operator d in the sequence

except the last, d must finish executing before the next operator in the sequence can start.

$\triangle$

Under these two assumptions, the total execution times for the two programs are reckoned as follows:

AlterV2 - There is only one such sequence of operators. The sum of execution times along it is

$$3S+4P+V$$

AlterS2 - There are two maximum-execution-time sequences:

$$S_1-S_2-C_3-U_2-S_4-F \text{ and } S_1-C_2-U_1-S_3-S_4-F$$

The total execution time along either is

$$S+4P+V$$

Thus the execution time for AlterS2 is 2S less than the time for AlterV2. Since this latter time is less than 8S, AlterS2 exhibits a reduction in execution time of at least 25%.

Thus the structure concurrency permitted in L_{BS} can result in significantly faster programs. Unfortunately, it can also result in non-functional programs, which are totally unacceptable in most computer applications. This inspires a search for a compromise, for a language in which it is easy to write programs which are functional but still exhibit as much of this concurrency as possible. L_{BV}, a rich language for the expression of functional algorithms over structured data, will be used as a paradigmatic source of functional programs.

Therefore, the primary goal of the thesis can be stated as:

Develop a language L_D, with interpreter, having the property
that any well-behaved L_{BV} program can be translated into an
equivalent L_D program which has maximal structure concurrency
(the L_{BV} program has none).

This involves four tasks:

1. Formally define what it means for certain programs in other
    languages to be equivalent to a given L_{BV} program (this is done
    in Section 2.4).

2. Develop L_D and its interpreter, as well as a translation to it
    from L_{BV} (Chapter 3).

3. Prove that the translation produces an equivalent program
    (Chapters 4, 5, 6, and 7).

4. Show that an L_D program on its interpreter is maximally concurrent.
    This is argued, though not proven, in Chapter 8.

2.4 Equivalence and Functionality

This section provides a formal definition of the equivalence of two
programs from different languages. This definition is not comprehensive,
but will cover the case of programs translated from L_{BV} to L_D. Since
every program in L_{BV} is functional, the equivalent L_D program must also
be functional. Therefore, a formal definition of functionality is given
first.

2.4.1 Functionality

An intuitive notion of a functional well-behaved program has been presented earlier: one which, given a set of input values, always necessarily produces the same set of output values. As has been pointed out, an initial (final) state establishes a set of program input (output) values. This suggests a more precise definition of functionality: Let S_1 and S_2 be any two initial states which establish the same set of input values. Then all halted firing sequences starting in S_1 and S_2 lead to final states which establish the same set of output values.

Two initial states for a program which establish the same program inputs can be characterized thusly: An arc has a token on it in either state iff it is a program input arc (because they are initial states); if an arc has tokens in the two states, then those tokens' values either are the same non-pointer value or are pointers to equal structures. With the substitution of "output" for "input", this same statement serves to characterize final states which establish the same set of output values. Both of these notions can be accomodated as special cases of the single more general concept of "equal states", developed next.

Two states of a program are equal iff, for each arc b in the program, the condition of b in one state matches that in the other; i.e., either

    1. b has no token in either state,

or 2. b has tokens of identical non-pointer values in the two states,

or 3. b has tokens whose values are pointers to equal components in the two states.

"Pointers to equal components in two states" need not be identical pointers, nor need they point to identical nodes. For example, let $S_1 = (\Gamma_1, (N_1, \Pi_1, SM_1))$ be an initial state, and let $S_2 = (\Gamma_1, (N_1, \Pi_2, SM_1))$ be a state identical to that except that some pointer p_1 in dom Π_1 is replaced in Π_2 by p_2 not in dom Π_1. Then certainly the component rooted at $\Pi_2(p_2)$ in S_2 is equal to the component rooted at $\Pi_1^{'}(p_1)$ in S_1. Therefore, the program has the same inputs in S_2 as in S_1.

Since nodes are only place-holders, uniformly substituting one for another in a heap does not change the data structure represented. For example, consider the initial state for AlterV depicted in Figure 2.3-2. The heap in that state is

$$U_1 = (N_1, \Pi_1, SM_1)$$

where $N_1 = \{m_1, m_2\}$

$\Pi_1 = \{(p_1, m_1), (p_2, m_2)\}$ where p_1 is the value of the token on the
X program input arc

$SM_1(m_1) = \{1, ('next', m_2)\}$

$SM_1(m_2) = \{2\}$

Replacing node m_2 with a different node $m_3 \neq m_1$ uniformly throughout U_1 yields

$$U_2 = (N_2, \Pi_2, SM_2)$$

where $N_2 = \{m_1, m_3\}$

$\Pi_2 = \{(p_1, m_1), (p_2, m_3)\}$

$SM_2(m_1) = \{1, ('next', m_3)\}$

$SM_2(m_3) = \{2\}$

The component of U_2 rooted at m_1 clearly represents the same structure as the component of U_1 rooted at m_1. Whenever two components are identical to within pointers and nodes, they are equal:

__Definition 2.4-1__ Let $U_1 = (N_1, \Pi_1, SM_1)$ and $U_2 = (N_2, \Pi_2, SM_2)$ be two heaps, and let I be any one-to-one mapping from N_1 to N_2. The component of U_2 rooted at any $m_2 \in N_2$ __equals__ __under__ I the component of U_1 rooted at $m_1 \in N_1$, written

$$U_2 \cdot m_2 \stackrel{I}{=} U_1 \cdot m_1$$

iff

1. $m_2 = I(m_1)$, and

2. for each $n \in N_1$ such that $n = m_1$ or n is reachable from m_1 in U_1,

   $$SM_2(I(n)) = I(SM_1(n))$$

where for any content $c = \{v, (s_1, n_1), \ldots, (s_j, n_j)\}$, $I(c)$ denotes the

   content $\{v, (s_1, I(n_1)), \ldots, (s_j, I(n_j))\}$.

$\triangle$

Now the definitions of matching conditions of arcs and of equal states follow directly:

__Definition 2.4-2__ Let $S_1 = (\Gamma_1, U_1)$ and $S_2 = (\Gamma_2, U_2)$ be two standard interpreter states, where $U_1 = (N_1, \Pi_1, SM_1)$ and $U_2 = (N_2, \Pi_2, SM_2)$. Let b_1 and b_2 each be an arc from the program of which Γ_1 and Γ_2, respectively, is a configuration. Then for any one-to-one mapping $I: N_1 \rightarrow N_2$, __the condition of__ b_2 __in__ S_2 __matches__ __under__ I __the__ __condition of__ b_1 __in__ S_1, written

$$\text{Match}((b_2, S_2), I, (b_1, S_1))$$

iff one of the following is true:

1. There is no token on b_1 in Γ_1 and none on b_2 in Γ_2.

2. There are tokens with the same non-pointer values on b_1 in Γ_1 and on b_2 in Γ_2.

3. There is a token with value $p_i \in V_p$ on b_i in Γ_i, $i=1,2$, and

$$U_2 \cdot \Pi_2(p_2) \overset{I}{=} U_1 \cdot \Pi_1(p_1)$$

$\triangle$

Definition 2.4-3 Let S_1 and S_2 be two standard interpreter states for the same data-flow program P. Then S_2 **equals** S_1 iff there is a single one-to-one mapping I under which, for every arc b in P, Match$((b,S_2)$, I, $(b,S_1))$.

$\triangle$

An initial state for a program P represents both P and a set of inputs for P. Equal initial states represent the same set of inputs for P. Similarly, equal final states represent the same set of program outputs for P (if P is well-behaved). For P to be functional, then, any two halted firing sequences starting in the same or equal initial states must yield the same or equal final states:

Definition 2.4-4 A program P is **functional** iff for every two equal initial states for P, S_1 and S_2, and halted firing sequences Ω_1 starting in S_1 and Ω_2 starting in S_2, $S_2 \cdot \Omega_2$ equals $S_1 \cdot \Omega_1$.

$\triangle$

Testing a program P for functionality according to this definition is a complex procedure: every initial state for P and every firing sequence starting in it must be checked. It is therefore worthwhile to seek ways to reduce this complexity; i.e., *a priori* conditions on two initial states S_1 and S_2 and firing sequences Ω_1 and Ω_2 starting in these states which will guarantee that $S_2 \cdot \Omega_2$ equals $S_1 \cdot \Omega_1$.

$S_2 \cdot \Omega_2$ equals $S_1 \cdot \Omega_1$ iff one can be obtained from the other by uniformly replacing certain distinct pointers and nodes with other distinct pointers and nodes. The pointers and nodes in any final state $S \cdot \Omega$ are the pointers and nodes in S plus those in the ordered pairs in the Copy, Const, Append, and Remove firings in Ω. Therefore, $S_2 \cdot \Omega_2$ equals $S_1 \cdot \Omega_1$ iff

1. S_2 is S_1 with certain pointers and nodes uniformly replaced with others, and

2. Ω_2 is Ω_1 with certain pointers and nodes in the ordered pairs in the firings replaced by others not in S_2.

The first condition has been formalized and abbreviated as "S_2 equals S_1". The second condition implies that the particualr pointer-node pairs in firings in a firing sequence have no bearing on the issue of functionality; that is, it is only the order of operator firings which matters. Removing the ordered pairs entirely from a firing sequence yields what may be termed its <u>reduction</u>; therefore, the second condition above is equivalent to "the reductions of Ω_1 and Ω_2 are identical." This may be further abbreviated as "Ω_2 equals Ω_1":

<u>Definition 2.4-5</u> Let Ω be any firing sequence. Then the <u>reduction</u> of Ω is obtained from Ω by replacing each firing $(d,(p,n))$ with the firing which is just d.

The reduction of any firing sequence starting in a state S is a <u>reduced firing sequence starting in</u> S.

Let Ω_1 and Ω_2 be any two firing sequences. Then Ω_2 <u>equals</u> Ω_1 iff the reductions of Ω_2 and Ω_1 are identical.

$\triangle$

Thus the complexity of testing for functionality is reduced by the fact that, if S_2 equals S_1 and Ω_2 equals Ω_1, then $S_2 \cdot \Omega_2$ equals $S_1 \cdot \Omega_1$ (Theorem 5.3-1).

Finally, a program together with a set of inputs to it can be associated with a class of initial states: the class of all those equal states which represent that program with those inputs. Any such class is in fact an equivalence class; i.e., the "equals" relation between states is an equivalence relation. This is proven as a corollary to the following:

Theorem 2.4-1 The "Match" relation is symmetric and transitive.

Proof: (The proof of this, which is a lengthy but straightforward manipulation of definitions, has been removed to Appendix A.)

$\triangle$

Corollary 2.4-1 The "equals" relation between states is an equivalence relation.

Proof: Reflexivity:

(1) Let $S = (\Gamma, U)$ be any state, where $U = (N, \Pi, SM)$. Let b be any arc in the program of which Γ is a configuration. Then either:

    b has no token in Γ and b has no token in Γ, or

    b has a non-pointer value in Γ and b has the same value, or

    b has a pointer value p in Γ and b has pointer value p, and

    $U.\Pi(p) \overset{I}{=} U.\Pi(p)$

        where I: $N \to N$ is the identity mapping   Def. 2.4-1

(2) There is a single map I: $N \to N$ under which, for each arc b,

    Match$((b, S),$ I, $(b, S))$ (1)+Def. 2.4-2

(3) S equals S (2)+Def. 2.4-3

Symmetry:

(4) Let S_1 and S_2 be two states for any program P. Then S_2 equals S_1

$\Rightarrow$ there is a single mapping I under which, for each arc b in P,

Match((b,S_2), I, (b,S_1))   Def. 2.4-3

(5) $\Rightarrow$ for each arc b in P, Match((b,S_1), I^{-1}, (b,S_2))   Thm. 2.4-1

(6) $\Rightarrow S_1$ equals S_2 Def. 2.4-3

Transitivity:

(7) Let S_1, S_2, and S_3 be three states for the same program P. Then S_2

equals S_1 and S_3 equals S_2 $\Rightarrow$ there are mappings I_1 and I_2 such

that, for each arc b in P, Match((b,S_2), I_1, (b,S_1)) and

Match((b,S_3), I_2, (b,S_2))   Def. 2.4-3

(8) $\Rightarrow$ for each arc b in P, Match((b,S_3), $I_2 \cdot I_1$, (b,S_1))   Thm. 2.4-1

(9) $\Rightarrow S_3$ equals S_1 Def. 2.4-3

2.4.2 Equivalence

The primary goal of the thesis is to develop a language L_D and an
interpreter for it such that an appropriate translation from L_{BV} to L_D
produces equivalent programs. The purpose of this section is to provide
a meaningful definition of equivalence of an L_D program to an L_{BV} program.
Intuitively, two programs are equivalent if both always produce the same
outputs from the same inputs. An initial state for either program
represents a set of inputs to that program. It is necessary now to
characterize two initial states which represent the same program inputs to
different programs.

Two initial states of the same program represent the same inputs iff
each program input arc has matching conditions in the two states.

Extending this to states of different programs requires first establishing
a one-to-one correspondence between their sets of program input arcs; then
it can be said that two initial states represent the same inputs iff the
conditions of corresponding program input arcs match. The analogous
characterization of equal program outputs necessitates a one-to-one
correspondence between sets of program output arcs in the programs.

When dealing with single programs in the discussion on functionality,
the notions of equal inputs and of equal outputs were generalized to the
concept of equal states: Two states of the same program are equal iff
every arc in the program has matching conditions in the two states.
Applying this generalization process to the case of two states of different
programs, however, is complicated by the possibility of syntactic differ-
ences between the programs. For example, substitutions like that shown in
Figure 2.2-3 will be used in the translation of an L_{BV} program to an L_D
program. Thus the latter will have extra arcs, such as that from the Copy
to the Assign in that Figure. Ignoring these arcs, however, there will be
an obvious one-to-one correspondence between the sets of remaining arcs in
the two programs. This is a similarity mapping:

Definition 2.4-6 Given two programs P and P', a similarity mapping A is
a one-to-one map from the arcs of P to the arcs of P' which carries
program input arcs to program input arcs and program output arcs to
program output arcs.

$\triangle$

Two states S and S' of different programs can be considered the same
if the conditions of at least the similar arcs in them match, In this
case, it will be said that S' simulates S:

<u>Definition 2.4-7</u> Let P and P' be two programs with a similarity mapping A from P to P'. Then a state S' of P' <u>simulates</u> a state S of P iff there is a single mapping I such that, for each arc b in P,

$$\text{Match}((A(b),S'),\ I,\ (b,S))$$

$\triangle$

A suitable formal definition of equivalence of programs in different languages follows directly from this:

<u>Definition 2.4-8</u> A program P' is <u>equivalent</u> to a program P iff:

1. There is a similarity mapping from P to P'.

2. For every initial state S of P and halted firing sequence Ω starting in S:

> for every initial state S' for P' simulating S, and halted
>
> firing sequence Ω' starting in S':
>
>> the final state $S'\cdot\Omega'$ simulates $S\cdot\Omega$.

$\triangle$

This definition is a weak one, but it is strong enough for the purpose of the thesis, which is again: To develop a language L_D, with interpreter, and a translation from L_{BV} to L_D which produces an equivalent program with maximal structure concurrency. Chapter 3 next develops the new language and interpreter, and the translation. Chapters 4, 5, 6, and 7 then prove formally that the translation does produce an equivalent L_D program from any well-behaved L_{BV} program.

Chapter 3

Controlling Structure Concurrency

This chapter contains the developments which meet the primary goal of the thesis. It describes a language L_D and an interpreter for it, designed so that every program is functional. It then gives an algorithm to translate any well-behaved L_{BV} program P into an equivalent L_D program P'. P' will exhibit a maximal degree of structure concurrency (subject to certain qualifications discussed in Section 8.2.1.4).

The chapter commences by studying structure concurrency in L_{BS} on the standard interpreter, to see exactly when it may cause non-functionality. Section 3.2 argues that this concurrency can be controlled so as to eliminate non-functionality through a combination of two techniques. The first is to re-write the program, inserting operators called sequencers at critical points. The second is to withhold the pointer-valued output tokens of a Select firing until certain existing tokens with the same pointer value have disappeared; this requires modifying the standard data-flow interpreter. The language L_D is just the set of L_{BS} programs which have sequencers in the right places and satisfy a restriction on the origins of pointer inputs to Assign, Update, and Delete operators. Every L_D program is functional on the modified data-flow interpreter. Section 3.4 then gives the translation algorithm, and proves that for every L_{BV} program P, its translation P' is in L_D, and that if P is well-behaved and P' is functional, then P' is equivalent to P; the proof that every L_D program is functional occupies the rest of the thesis.

3.1 Interference

It is a well-established principle [9] that the key to guaranteeing functionality is preventing interference:

Definition 3.1-1 Given an initial state S for any data-flow program P and a firing sequence $\Omega\varphi_1\varphi_2$ starting in S, the two firings φ_1 and φ_2 **interfere** (**with each other**) iff:

1. $\Omega\varphi_2\varphi_1$ is also a firing sequence starting in S, and

2. $S\cdot\Omega\varphi_1\varphi_2$ and $S\cdot\Omega\varphi_2\varphi_1$ are not identical states.

$\triangle$

An example of interference can be seen between firings of the actors U and S_2 in AlterS. Figure 2.3-4 depicts an initial state S for AlterS. One firing sequence starting in S is $\Omega = G,S_1,C_1,C_2$. In the state $S\cdot\Omega$ (depicted in Figure 2.3-6), both S_2 and U are enabled. Therefore, both Ω,S_2,U and Ω,U,S_2 are firing sequences starting in S (since data flow is **persistent**: firing one actor cannot disable another). However, the states $S\cdot\Omega,S_2,U$ (Figure 3.1-1(a)) and $S\cdot\Omega,U,S_2$ (Figure 3.1-1(b)) are clearly not identical: the tokens on the output arcs of S_2 have as values pointers to different nodes.

The reason for this interference is that firing Update U changes the ordered pair with selector 'next' in node n_1's content, while firing Select S_2 reads the ordered pair with selector 'next' in that content. Therefore, S_2 may or may not read the pair written by U, depending on whether or not U fires before it. Because of this dependence, the existence of two firing sequences in which S_2 and U fire in a different relative order implies the possibility of non-functionality. This interference does not imply the necessity of non-functionality; there are pathological cases, discussed

(a)

$S \cdot G, S_1, C_1, C_2, S_2, U$

(b)

$S \cdot G, S_1, C_1, C_2, U, S_2$
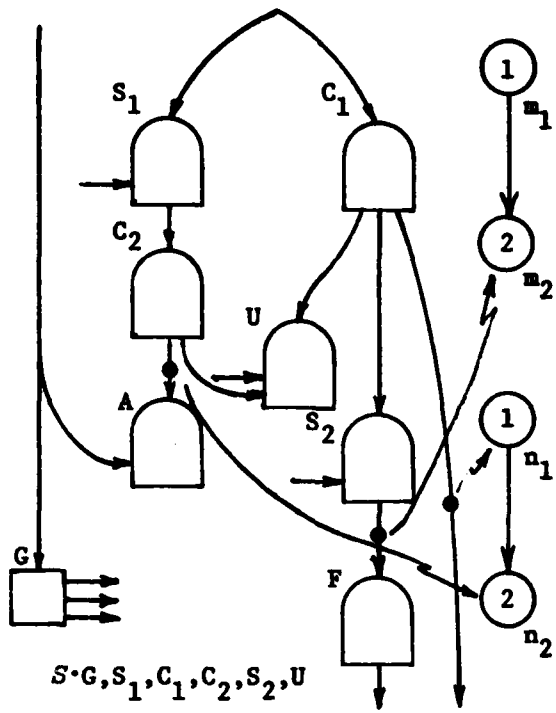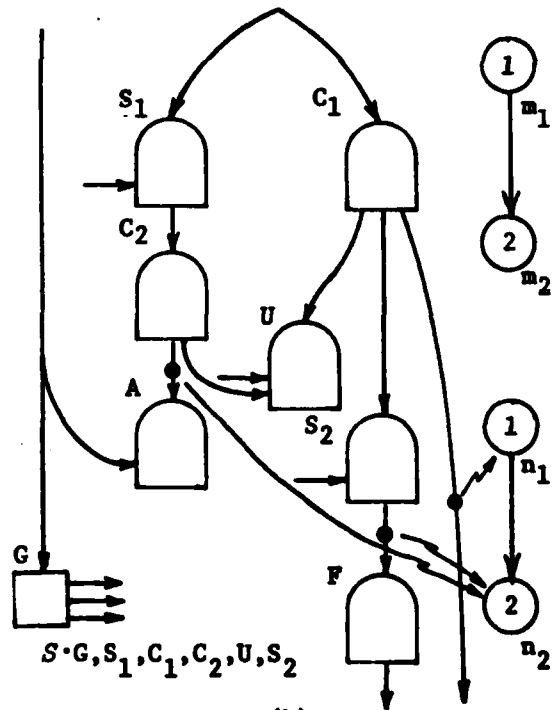
An Example of Interference

Figure 3.1-1

shortly, in which it is harmless. But eliminating all possible interference is the easiest way to guarantee functionality. For this reason, it is important to be able to recognize all potential instances of interference.

The interference between U and S_2 falls into the broad category of one operator trying to change a stored item and another trying to read that item. (An <u>item</u> is either the value in a given node's content or the ordered pair in that content containing a given selector.) Interference can also occur between two operators trying to change the same stored item. For example, if two Assign operators d_1 and d_2 are both enabled in some state $S \cdot \Omega$ with pointers to a node n as inputs, then in either of the states $S \cdot \Omega d_1 d_2$ or $S \cdot \Omega d_2 d_1$, the value of n is the value stored by the last of d_1 and d_2 to fire. Thus these two states are different (except in the singular case that both Assign firings wrote the same value). Such interference may lead to non-functionality in one of two ways:

1. An fetch of n's value immediately after the last of these Assign firings will have different outputs in different firing sequences.

2. If no Assign firing follows these two, then different firing sequences lead to final states in which n has different values.

The strategy for guaranteeing functionality, presented in Section 3.2, is to eliminate all potential interference. As a first step, the above generalizations are particularized to L_{BS}, yielding exact conditions under which two firings potentially interfere.

3.1.1 Potential Interference in L_{BS}

This section analyzes the conditions under which firings φ_1 and φ_2 of actors d_1 and d_2 in a program P can interfere: what types of actors d_1 and

d_2 must be and how the firings' inputs must be related. It assumes the requisite initial state S for P and firing sequence $\Omega\varphi_1\varphi_2$ starting in such that $\Omega\varphi_2\varphi_1$ is also a firing sequence starting in S.

Both d_1 and d_2 must be enabled in state $S\cdot\Omega$. This means that each of them has tokens on all of its input arcs in that state. It is the fortunate property of data flow that nothing can change the value of a token once it appears on an arc. Therefore, the values input by firings φ_1 and φ_2 do not depend on which firing occurs first.

If one of these actors, say d_1, is not a structure operator, then the only effect of φ_1 on the state is to place tokens on d_1's output arcs. The values of those tokens depend only on the values input by φ_1. Therefore, the state change effected by φ_1 does not depend on whether or not it precedes φ_2. Since the effect of φ_2 depends on at most φ_2's inputs and the heap, neither of which is changed by φ_1, it is independent of whether or not φ_2 follows φ_1.

Therefore, for φ_1 and φ_2 to interfere, d_1 and d_2 must both be structure operators. As noted in general earlier, one of the firings, say φ_2, must change a stored item, and the other must either change that same item or output a value which depends on that item. Each firing of a structure operator changes or depends upon items wholly within one node's content. That node is either the one pointed to by the firing's input or one activated by the firing. If φ_2 were to change the content of a node n which it activated, then φ_1 could not change or depend upon n's content. This is because (1) node n is not active in $S\cdot\Omega$, before φ_2, and so no token in that state, including φ_1's input, has a pointer to n as its value, and (2) if φ_1 also activates a node, then that node is

necessarily distinct from n. Therefore, φ_2 must change the content of the node pointed to by its input, and φ_1 must input this same pointer. This implies that d_2 must be either an Assign, Update, or Delete operator. (Since no L_{BV} operator can change the content of an already-active node, no two firings of L_{BV} operators can interfere. Hence, all L_{BV} programs are functional.)

From this, necessary conditions for firings φ_1 and φ_2 to interfere include:

1. both are firings of structure operators, one an Assign, Update, or Delete, and

2. both have the same (number-1) pointer input.

Certain pairs of firings cannot interfere, by virtue of the actions of the actors of which they are firings. Otherwise, the firings _potentially interfere_. The strategy for precluding actual interference in this latter case is to insure that the firings are _sequenced_ by S; i.e., that they occur in the same relative order in all firing sequences starting in S. This strategy is explained and justified shortly. First, the structure operations in L_{BS} are examined to see which ones cannot potentially interfere with each other.

The following observations are made about potential interference between two firings of L_{BS} structure operators with pointer inputs (number-1 pointer input for an Update) equal to p: Let $m = \Pi(p)$.

1. The L_{BS} structure operators can be partitioned into two classes:

    read class - Fetch, First, Next, Select, Copy

    write class - Assign, Update, Delete

    so that no two firings of read-class operators potentially interfere.

2. A Copy firing potentially interferes with any firing of a write-class operator, because the content of the node activated by the Copy firing depends on the entire content of m.

3. An Assign firing potentially interferes with a Fetch or Assign firing.

4. A First or Next firing potentially interferes with an Update or Delete firing, because. the latter modifies the set $O(m)$ of selectors off m. This has the following possible consequence: Let s_1 be the selector input of a Next firing, or in the case of a First firing, let s_1 be a lower bound of the selector domain Σ with respect to the total ordering $<$. Let s_2 be the selector which would be output by firing the First or Next first. Then either:

   a. a Delete firing could remove the ordered pair with s_2, or

   b. an Update firing could add a pair with selector s_3 such that

$$s_1 < s_3 < s_2$$

5. A Select firing potentially interferes with an Update or Delete firing iff their selector inputs are the same.

6. Two Update firings or an Update and a Delete firing potentially interfere iff their selector inputs are the same.

These observations are summarized in the following:

Definition 3.1-2 The read class of L_{BS} structure operations consists of Fetch, First, Next, Select, and Copy. The write class consists of Assign, Update, and Delete.

Given an initial state S for an L_{BS} program, and a firing sequence Ω starting in S, any two firings in Ω having the same number-1 input

potentially interfere based on their operations and selector inputs, according to Table 3.1-1.

$\triangle$

Theorem 3.1-1 Two firings interfere only if they potentially interfere.

$\triangle$

3.1.2 Determinacy

Given an initial state S for a program P, the existence of two potentially-interfering firings φ_1 and φ_2 in a firing sequence starting in S does not necessarily imply that P is non-functional. P may still be functional for any of the following reasons:

1. (Sequencing) There is no firing sequence Ω starting in S such that both $\Omega\varphi_1\varphi_2$ and $\Omega\varphi_2\varphi_1$ are firing sequences starting in S.

2. (Repetition) There is such an Ω, but $S\cdot\Omega\varphi_1\varphi_2$ and $S\cdot\Omega\varphi_2\varphi_1$ are identical states. This may occur if, for example, φ_2 is an Assign firing which assigns v as the value of node n, and either:

    a. φ_1 is a Fetch firing and n has value v in $S\cdot\Omega$, so that φ_1 outputs v whether or not φ_2 precedes it, or

    b. φ_1 is an Assign firing which also assigns value v, so that n has value v after both firings regardless of their order.

3. (Lossiness) φ_1 and φ_2 actually interfere, but the aspects in which states $S\cdot\Omega\varphi_1\varphi_2$ and $S\cdot\Omega\varphi_2\varphi_1$ differ do not enter into determination of the final state.

Potentially-interfering firings in a firing sequence are inevitable, for there is little purpose in a firing which writes an item into a node's content if no subsequent firing will ever read that item. Insuring a program's functionality thus necessitates (1) identifying whenever firings

	Fetch	First	Next	Select	Copy	Assign	Update	Delete
Fetch						√		
First							√	√
Next							√	√
Select							*	*
Copy						√	√	√
Assign	√				√	√		
Update		√	√	*	√		*	*
Delete		√	√	*	√		*	

Legend:

    √ two firings of operators of these types potentially interfere

    * two firings of operators of these types potentially interfere
        iff their selector inputs are the same

Interference Potential of Firings with the Same Pointer Input

Table 3.1-1

of two operators potentially interfere, and (2) seeing that each such in-stance does not induce non-functionality, for one of the above three reasons.

By far the most common reason for functionality is freedom from conflict. A conflict-free program is one in which every pair of potentially-interfering firings is sequenced:

Definition 3.1-3 Given any initial state S for a data-flow program P, the ith firing of actor d_1 in P is sequenced by S after the jth firing of

<u>actor</u> $\underline{d}_2$ iff, for all firing sequences Ω starting in S, the ith firing of d_1 in Ω follows the jth firing of d_2 in Ω.

A program P is <u>conflict-free</u> iff the following is true for every initial state S for P and every two structure operators d_1 and d_2 in P: If there is a firing sequence starting in S in which the ith firing of d_1 potentially interferes with the jth firing of d_2, which it follows, then the ith firing of d_1 is sequenced by S after the jth firing of d_2.

$\triangle$

Functionality of a conflict-laden program by virtue of repetition or lossiness is pathological and difficult to verify. Therefore, the strategy for guaranteeing functionality is to guarantee freedom from conflict.

Lack of conflict in fact implies a much stronger property of programs than functionality: <u>determinacy</u>. A determinate program is one which not only always produces the same outputs given the same inputs, but always does so "in the same way". This important concept is made more precise in the following:

Given a program P, the set of all initial states which represent the same inputs to P is an equivalence class E. Therefore, P is determinate iff any two firing sequences Ω_1 and Ω_2 starting in any two states in any such E lead "in the same way" to equal final states, where "in the same way" is defined by the five <u>Determinacy Assertions</u> discussed in the following paragraphs.

The sets of firings in Ω_1 and Ω_2 must be the same, and each firing must have the same set of non-pointer inputs in Ω_1 and Ω_2:

1. For each actor d in P, the number of firings of d in Ω_1 equals the number of firings of d in Ω_2.
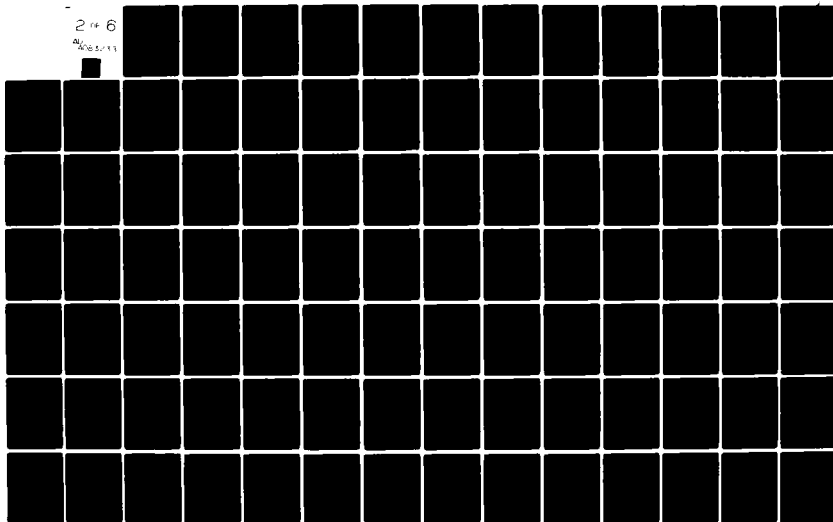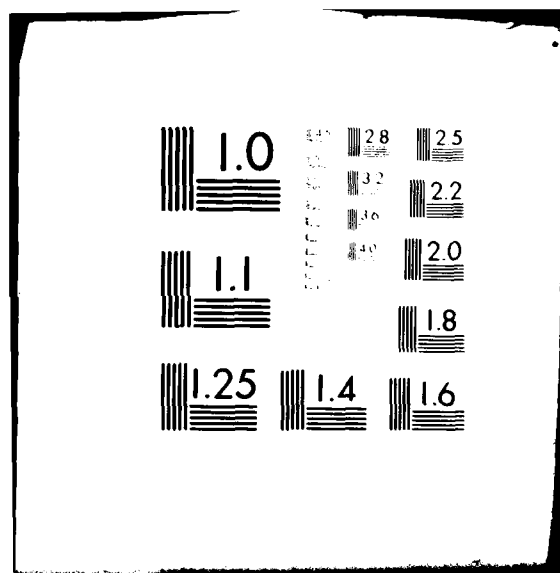
2. For any actor d and integers i and j, the number-i input to the jth firing of d in Ω_1 is not a pointer iff the number-i input to the jth firing of d in Ω_2 is not a pointer. Furthermore, if those two input values are not pointers, then they are identical.

Since pointer values are arbitrary, any single given firing may have different pointer-valued inputs in Ω_1 and Ω_2. If two different firings both have the same pointer-valued input in Ω_1, however, then those firings must have the same pointer-valued input in Ω_2. Put another way:

3. There is a one-to-one map F over pointers such that the number-i input to the jth firing of d in Ω_1 is pointer p iff the number-i input to the jth firing of d in Ω_2 is F(p).

In addition to these constraints on the value of an input to a firing, the other firing from whose output that value was transferred must be the same in Ω_1 and Ω_2, whether the transfer is direct or indirect. A direct transfer occurs via an arc of the program: if the token removed from an input arc of actor d_1 by its jth firing was placed there by the kth firing of actor d_2, then the value of that token was transferred directly from the latter output to the former input. Thus:

4. For every arc b in P, let b be an output arc of d_2 and an input arc of d_1. Then the jth firing of d_1 in Ω_1 removes a token placed on b by the kth firing of d_2 iff the jth firing of d_1 in Ω_2 removes a token placed on b by the kth firing of d_2.

An indirect transfer of value v from the kth firing of d_2 to the number-i input of the jth firing of d_1 proceeds via the heap, as follows: First v is transferred directly from the kth firing of d_2 to the number-2 input of an Assign firing, which writes v as the value of a node.

Subsequently, v is output by a Fetch firing F which is in the reach of A. Then it is transferred directly from F's output to the number-i input of the jth firing of d_1. The complex concept of reach is central to the understanding of the interrelationships between firings of structure operators. It is discussed in detail in Chapter 5; since the intent here is only to provide an intuitive introduction to determinacy, the following brief explanation of reach should suffice.

Let A be any Assign firing which inputs a pointer p, and let $n = \Pi(p)$. For any other Fetch or Assign firing F in the same firing sequence, let q be F's pointer input and let $m = \Pi(q)$. Then F is in the reach of A iff its outputs necessarily depend just on the value written by A; i.e., iff

a. $m = n$ and F occurs while n still has the value written by A, or

b.   m is a copy of n made while n still had the value written by A, and
F occurs while m still has its initial value copied from n.

The indirect transfers will be the same in Ω_1 and Ω_2 if the direct transfers are the same and

5.   For each Assign (or Update/Delete) firing A, the reach of A in Ω_1 contains the same firings as the reach of A in Ω_2.

The five assertions just listed complete the definition of a determinate program. The awkward statement of the definition in terms of the usual model of data flow is a major motivation for the development, in Chapter 4, of a new model of concurrent computation. This model permits a more precise definition of reach and a much more concise definition of determinacy, as will be seen in Chapters 5 and 6.

Since determinacy is the only practical path to functionality, the primary goal of the thesis is refined thusly: Develop a language L_D and an interpreter for it, together with a translation algorithm which takes any well-behaved L_{BV} program P into an L_D program which, on the new interpreter, is determinate, equivalent to P, and maximally-concurrent. This development is in three steps:

1. Modify the standard interpreter so that an easily-recognized class L_D of L_{BS} programs are conflict-free and have maximal concurrency consistent with that freedom.

2. Prove that freedom from conflict guarantees determinacy of a data-flow program, and that determinacy in turn guarantees functionality.

3. Present a translation algorithm which takes any well-behaved L_{BV} program into an equivalent L_D program.

The first and third steps are undertaken in the remainder of this chapter; the second step occupies the rest of the thesis.

3.2 Guaranteeing Determinacy

This section describes techniques for eliminating enough structure concurrency from an arbitrary L_{BS} program to guarantee its determinacy. Any data-flow program without structure operators is necessarily determinate. The presence of structure operators inposes the following additional requirement: If two firings potentially interfere in any firing sequence starting in initial state S, then they must be sequenced by S.

The easiest way to sequence the ith firing of d_1 after the jth firing of d_2 is to ensure that d_1 is not enabled for the ith time until after d_2 has fired for the jth time. The easiest way to prevent an actor's being enabled is to deny it one of its input tokens. The only input common to all structure operators is a number-1 pointer input. Accordingly, techniques are presented in this section to:

1. identify which firings of which structure operators in a program might potentially interfere in a firing sequence Ω, and

2. sequence each such pair of firings, by withholding the pointer input to the second until the first has occurred.

There are two different techniques used, depending on whether the two firings are in the same <u>blocking group</u> in Ω or in different ones. Section 3.2.1 defines blocking groups and explains why different techniques are appropriate in the two cases. Sections 3.2.2 and 3.2.3 then describe the two techniques for identifying and sequencing potentially-interfering firings.

3.2.1 Blocking Groups

Every pointer-valued token appearing in a state can be traced back to a unique origin, either a program input token or the output of a firing of a Copy or Select operator. (The only other way in which a pointer-valued token appears on an arc is as the output of a firing of a pI actor which removed an identical token from another arc; such firings are thought of as propagating rather than creating the token.) A blocking group in a firing sequence consists of all firings which remove pointer-valued tokens having a common origin. The origin of each token is easily perceived by considering the tagged data-flow interpreter, explained below in Section 3.2.1.1. Section 3.2.1.2 then explains the significance of two firings being in the same or in different blocking groups.

3.2.1.1 The Tagged Interpreter

The tagged data-flow interpreter is introduced informally here, purely for explanatory purposes. It is not the modified interpreter which meets the goal of the thesis; that is defined formally later.

A state of the tagged interpreter is similar to a standard inter-preter state (Definition 2.1-3). The only difference is that pointers are replaced by tagged pointers as values for tokens. A tagged pointer is an ordered pair consisting of a pointer p and a tag e, which indicates the origin of the token. This tagged pointer is written TP(p,e).

An initial state of the tagged interpreter is one obtained from an initial standard state by giving each pointer-valued token a new value, as follows: If the token is on the number-i input arc of the program and has value p, then its new value is TP(p,Tg(ID,i)). Each firing of a pI

actor which inputs a token with a tagged pointer outputs a token with the same tagged pointer. A firing of a structure operator with input $TP(p,e)$ ignores the tag e, outputting, in general, the same values as it would on the standard interpreter given just p as input. The exception is that the nth firing of a Copy or Select operator d in a firing sequence outputs tagged pointers with the tag $Tg(d,n)$.

There is a one-to-one correspondence between the possible sequences of states undergone by the standard interpreter and those undergone by the tagged interpreter. For example, for any standard state sequence starting in an initial state for program AlterS, the corresponding tagged state sequence is given by the following algorithm:

Replace any token with pointer value p appearing on	with a token having as value the tagged pointer
the X program input arc	$TP(p,Tg(ID,2))$
the output arc of S_i, $i=1,2$	$TP(p,Tg(S_i,1))$
the output arc of C_i, $i=1,2$	$TP(p,Tg(C_i,1))$

Therefore, a program is determinate on the tagged interpreter iff it is determinate on the standard interpreter.

3.2.1.2 Intra-Group versus Inter-Group Sequencing

Each firing of a structure operator on the tagged interpreter removes a token having a tagged pointer as value. The tag identifies the origin of the token. For each tag e and firing sequence Ω, the <u>blocking group</u> $B_\Omega(e)$ is the set of all firings in Ω which remove tokens with tag e from their number-1 input arcs (or more precisely, which remove tokens with which are associated tagged pointers containing tag e). For example, in

any firing sequence Ω for AlterS, the firings of U and S_2 are both in $B_\Omega(Tg(C_1,1))$, while the firings of A and F are in distinct blocking groups $B_\Omega(Tg(C_2,1))$ and $B_\Omega(Tg(S_2,1))$.

Two tokens with the same tag have the same pointer value. Thus, two structure operator firings in the same blocking group necessarily have equal number-1 inputs. Their interference potential is then determined solely by their operations and their selector inputs, according to Table 3.1-1. The firings of U and S_2, which are always in the same blocking group, always potentially interfere. Since they are not sequenced by any initial state, AlterS is non-determinate. It is easy to ascertain syntactically whether or not firings of two actors in a program can ever be in the same blocking group; this is done in Section 3.2.1.3. If so, the program can be re-written to guarantee that one of the actors is never enabled with a given tagged pointer as input until the other one has consumed an identical input. This is demonstrated in Section 3.2.2.

Two firings in distinct blocking groups may or may not have equal pointer inputs. From Figure 3.1-1, the firings of A and F in AlterS will have equal pointer inputs if S_2 fires after U. I.e., A and F potentially interfere, even if S_2's firing is sequenced after U's (Figure 3.2-1). Thus, AlterS will be determinate only if A and F are somehow sequenced. But in a similar program in which U and S_2 could have unequal selector inputs, A and F could have unequal pointer inputs. In that case, A and F should not be sequenced, in the interest of increased concurrency. Therefore, firings in distinct blocking groups should be sequenced only if their pointer inputs are actually equal. In general, this discrimination is
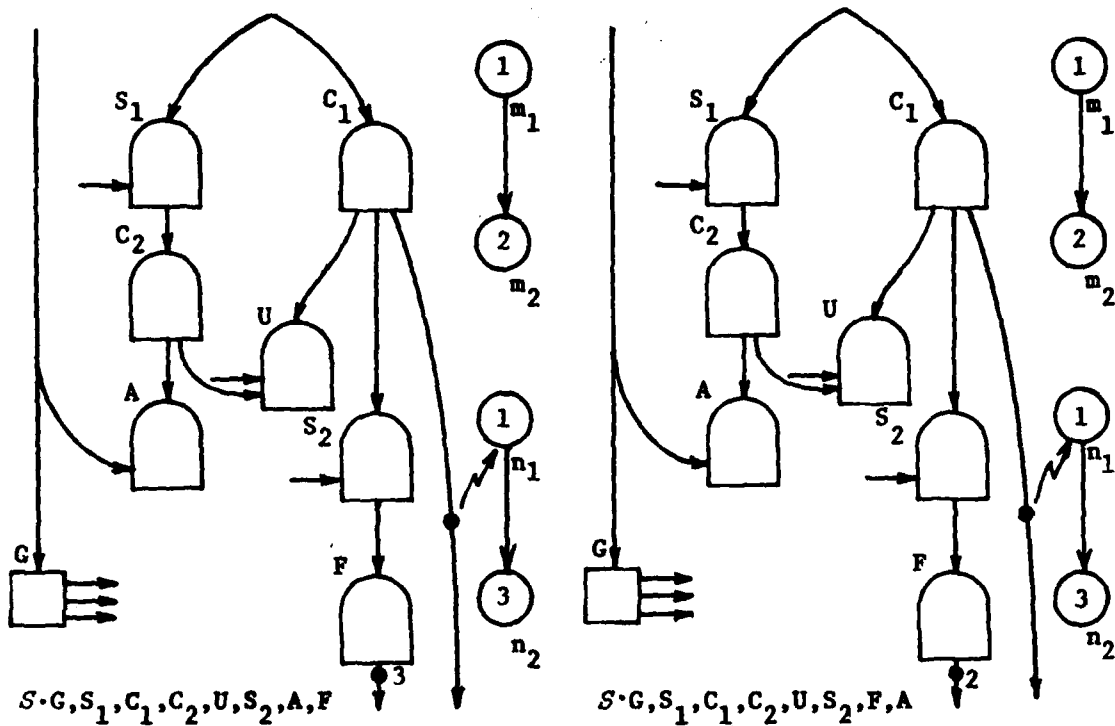
A Further Example of Interference

Figure 3.2-1

possible only through a "run-time" inspection of these inputs. This requires modifying the interpreter, as described in Section 3.2.3.

3.2.1.3 Distribution Groups

As mentioned, it is easy to identify those pairs of actors in a program of which firings can be in the same blocking group: Firings of two actors can be in the same blocking group only if the actors are in the same maximal pointer distribution group, defined in the following.

Definition 3.2-1 A kernel in a program P is a subset K of the data arcs of P which satisfies one of the following two specifications:

1. For any i, let b be the number-i program input arc of P. Then {b} is a kernel, the one denoted K(ID,i).

2. For any actor d in P, and for i = 1 or i = 2, the set

   {b| b is in the number-i group of output arcs of d} is a kernel,

   and is denoted K(d,i).

The primary input arc of a structure operator is its number-1 input arc; the primary input arcs of a pI actor are its transmitted-input arcs.

For any arc b in P, the channels starting at b are subsets of the data arcs of P satisfying the following recursive specification:

1. b is in every channel starting in b.

2. If b is an output arc of a pI actor d, then any channel

   containing a primary input arc of d also contains arc b.

For any kernel K, the distribution group for K, G(K), is the set of all structure and pI actors in P whose primary input arcs are in channels starting at arcs in K.

The set of maximal pointer distribution groups (m.p.d.g.'s) in P is

$\{G(K)| \exists i: K = K(ID,i) \vee$

$\exists S: S$ labels a Select operator in P and $K = K(S,1) \vee$

$\exists C: C$ labels a Copy operator in P and $K = K(C,1)$ or $K = K(C,2)\}$

$\triangle$

An m.p.d.g. describes a relation among the actors in a program which is static, based only on the unchanging program. A blocking group, on the other hand, establishes a dynamic relation among firings of actors, which may change from one firing sequence to another. The two relations are closely coupled, as shown in

The <u>Static/Dynamic</u> <u>Group</u> <u>Relationship</u>: Given a firing sequence Ω for

program P on the tagged interpreter, for every firing φ in Ω of a

structure operator d in P, there is a tag e such that $\varphi \in B_\Omega(e)$. Also:

1. If $e = Tg(ID,i)$ for some i, then d is in $G(K(ID,i))$.

2. If $e = Tg(S,n)$ for some n, where S is a Select or Copy operator in

    P, then d is in $G(K(S,1))$, if S is a Select, or d is in $G(K(S,1))$

    or $G(K(S,2))$, if S is a Copy.

According to this relationship, which is proven as Lemma 3.3-1,

firings of each of two actors are in the same blocking group only if the

actors are in the same or closely-related m.p.d.g.'s. E.g., in AlterS the

firings of U and S_2 are always in a common blocking group, and U and S_2

are in the same m.p.d.g. (The reason for the ungainly separation into two

m.p.d.g.'s per Copy operator will be explained at the end of Section 3.3.)

Not all firings of actors in one m.p.d.g. are in the same blocking

group. For example, if AlterS were embedded in a loop, so that each actor

in it fired several times in one firing sequence, then for each n, the only

firings to input tokens with tag $Tg(C_1,n)$ would be the nth firings of U

and S_2. Thus the ith firing of U and the jth firing of S_2, for $i \neq j$, would

be in distinct blocking groups.

With this background, it is easy to explain the two techniques for

identifying and sequencing potentially-interfering firings: one for firings

in the same blocking group and the other for distinct groups.

3.2.2 Sequencing Within a Blocking Group

This section introduces the technique for identifying and sequencing

every pair of potentially-interfering firings which are in the same

blocking group. The identification problem has already largely been solved: A firing of actor d_1 in program P is not in the same blocking group as a firing of d_2 unless d_1 and d_2 are in the same m.p.d.g. Furthermore, Table 3.1-1 may show that firings of d_1 and d_2 could never potentially interfere. Otherwise, certain pairs of firings of d_1 and d_2 will be in common blocking groups and will potentially interfere. Those pairs, in which on the tagged interpreter both firings remove tokens with the same tag, must be sequenced by every initial state of P.

The sequencing problem is to guarantee that a certain firing of d_2, say the jth, which removes a token with tag e, never occurs until after, say, the ith firing of d_1, which removes an identical token. As mentioned earlier, the surest solution is to prevent the jth appearance of a token on d_2's input arc until after the ith firing of d_1 has occurred. This can always be accomplished by re-writing the program, inserting a sequencer between d_1 and d_2:

<u>Definition 3.2-2</u> An (<u>r-ary</u>) <u>sequencer</u> is an r-ary data-flow operator with which is associated the projection function P_1^r, defined by

$$P_1^r(x_1, x_2, \ldots, x_r) = x_1$$

$\triangle$

Just like any other data-flow operator, a sequencer is not enabled to fire until it has tokens on all of its input arcs. When it fires, it ignores all but one input token, and places on its output arcs tokens identical to that one input token (which on the tagged interpreter may be a tagged pointer). Therefore, a sequencer is a pI actor, with its number-1 input arc being its only transmitted-input arc.

The program AlterS' (Figure 3.2-2) illustrates the use of a sequencer to sequence all potentially-interfering firings of U and S_2 in AlterS. (This figure uses a graphical convention in which the transmitted-input arc of a sequencer is connected to its output arcs through the actor symbol.) For each n, the nth firing of Copy C_1 on the tagged interpreter places tokens with the unique tag $e = Tg(C_1,n)$ on all output arcs of C_1. Those tokens will be input by the nth firings of U and of sequencer Q. Tokens with tag e cannot appear on other arcs of the program until after the nth firing of Q. Q is not enabled for the nth time until the nth appearance of some token on its other input arc. This appearance occurs as a result of the nth firing of U, which is the firing of U which consumes a token with tag e. Therefore, no tokens with tag e can appear on the input arcs of any structure operators other than U before that firing of U which consumes a token with tag e. Of all the firings of structure operators in blocking group $B_\Omega(e)$, the first to occur is the firing of U. Therefore, those firings of U and S_2 which are in the same blocking group are sequenced.

Sequencers can be used in this manner to sequence any two firings in the same blocking group. Any program in which, for every initial state S, every pair of potentially-interfering firings in a common blocking group are sequenced by S satisfies the Determinacy Condition (this statement will be formalized later.)

The example of AlterS' suggests a simplistic algorithm to translate any L_{BV} program into an L_{BS} program:
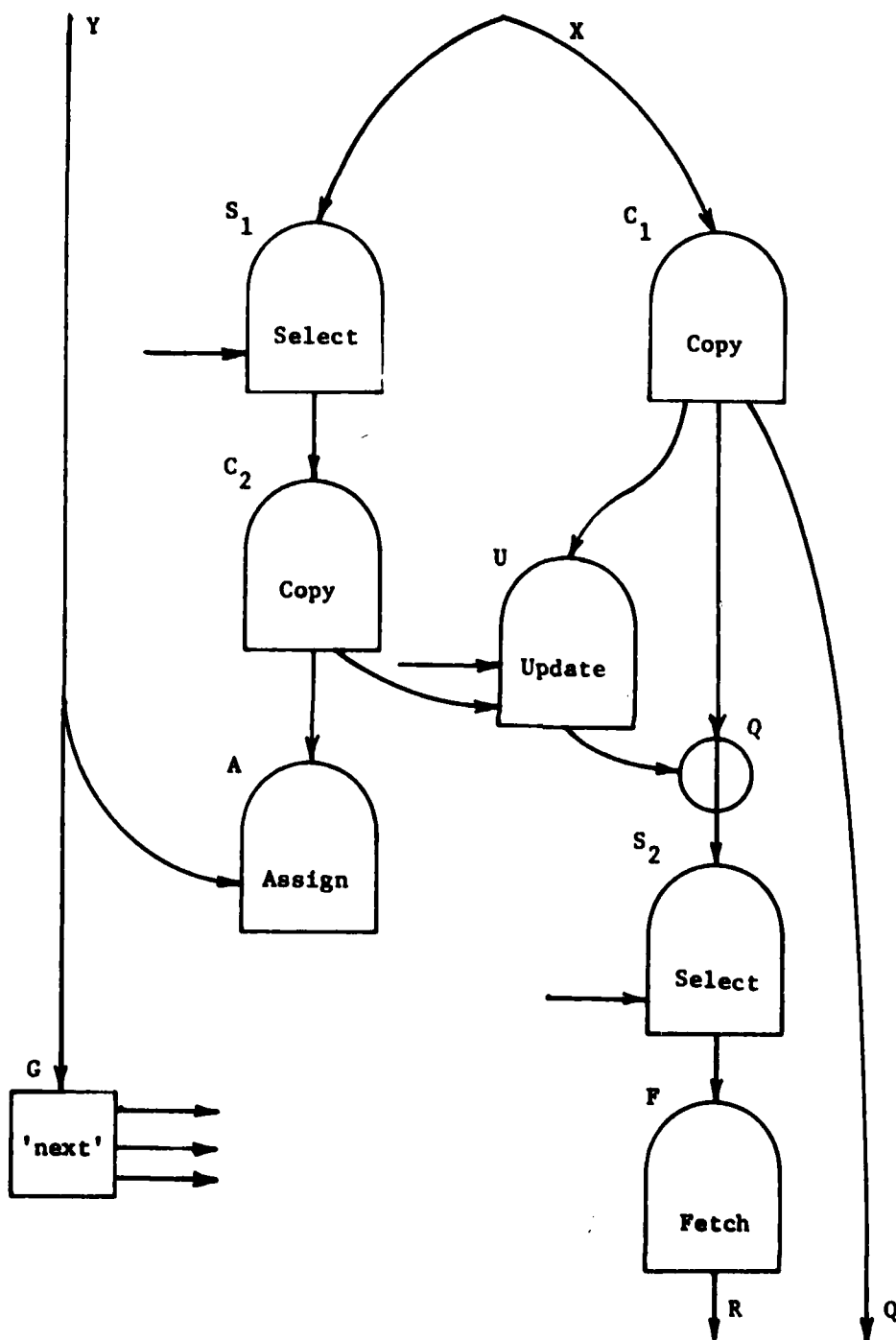
The Program AlterS'

Figure 3.2-2

<u>Algorithm 3.2-1</u> For any L_{BV} program P, perform the substitutions shown in Figure 3.2-3 for every Const, Append, and Remove operator in P.

$\triangle$

Denote by L_S the set of L_{BS} programs

$L_S = \{P' \mid P'$ is the translation of an L_{BV} program$\}$

Then the following argues informally that each P' in L_S satisfies the Determinacy Condition; a formal proof is given in Section 3.4.

For any L_{BV} program P, let P' be the program resulting from applying Algorithm 3.2-1 to P. Let d_1 and d_2 be two operators in P' such that there is a firing sequence Ω in which two firings of d_1 and d_2 are in the same blocking group $B_\Omega(e)$ and potentially interfere. Then, from Table 3.1-1, at least one of the actors, say d_1, must be a write-class operator: Assign, Update, or Delete. Since there is no such operator in any L_{BV} program, d_1 must have been introduced into P' by the translation algorithm. Comparing Figures 3.2-2 and 3.2-3, then, in P' the operator d_1 and a sequencer are connected to each other and to the outputs of a Copy operator just as are U and Q in AlterS'. Therefore, the conclusion drawn with respect to the latter program applies to the former: Every firing of d_1 is sequenced before any other structure operator firing which is in the same blocking group. Therefore, any two firings which potentially interfere and are in the same blocking group are sequenced. This is the Determinacy Condition, and it holds for every program P' which is the translation of any L_{BV} program.

This completes the informal explantion of how, at least in L_S programs, all pairs of potentially-interfering firings in common blocking
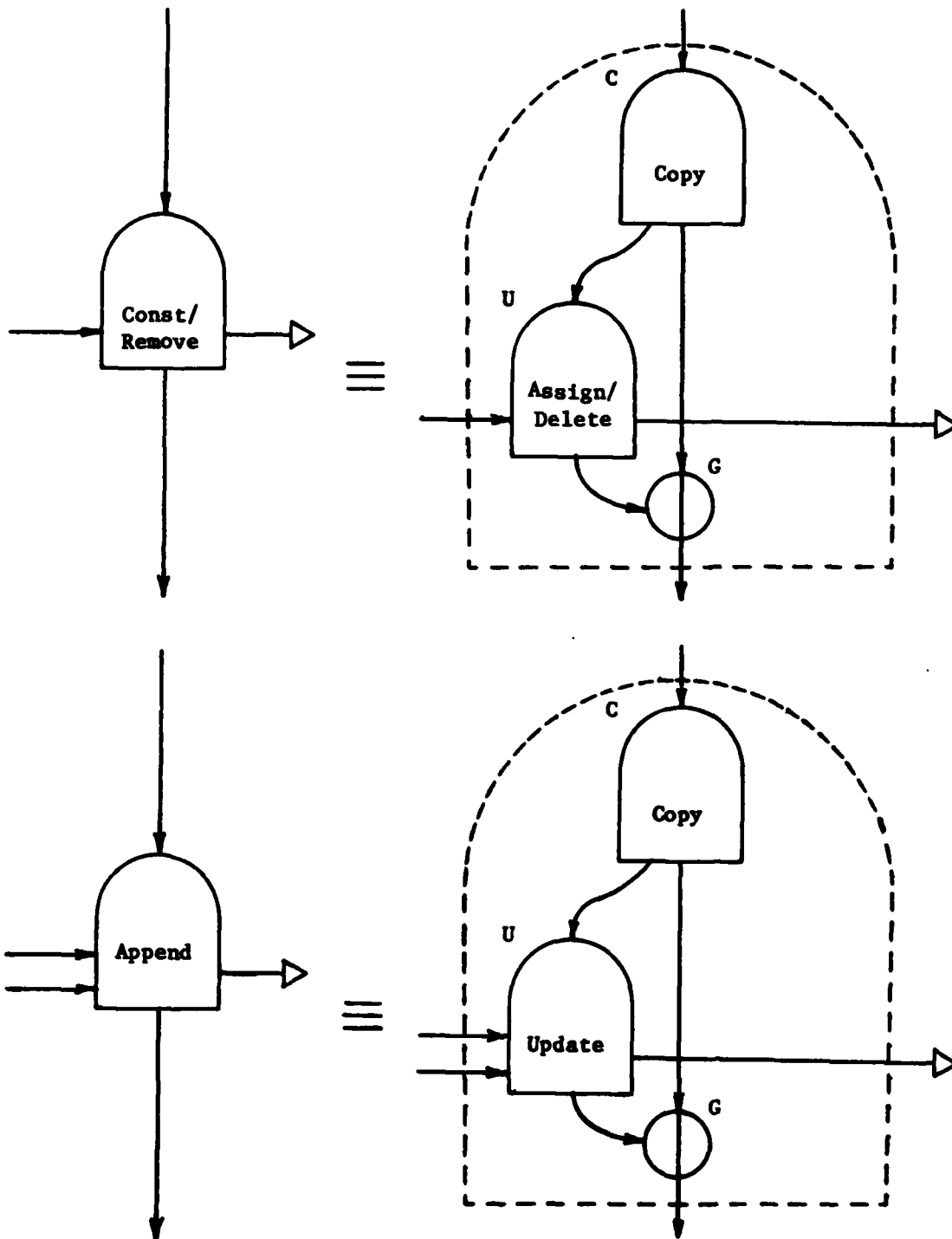
Operator Substitutions in Translating from L_{BV} to L_S

Figure 3.2-3

groups are sequenced. The technique is formally verified in Section 3.4. The next sub-section now describes a new technique for sequencing pairs of firings from distinct blocking groups, which is a major contribution of the thesis.

3.2.3 Sequencing Firings in Distinct Blocking Groups

This section analyzes the programs in L_S with the aid of the tagged interpreter. The goal is a method which insures the following for any firings sequence Ω starting in any initial state S: If two firings which are in different blocking groups in Ω potentially interfere, then they are sequenced by S. I.e., for any two distinct blocking groups $B_\Omega(e_1)$ and $B_\Omega(e_2)$, the method must (1) determine which pairs of firings, one from each group, potentially interfere, and then (2) insure that each such pair is sequenced.

The most straightforward method will be used, which is to:

(1) determine if there is <u>any</u> firing in $B_\Omega(e_1)$ which potentially interferes with <u>any</u> firing in $B_\Omega(e_2)$, and

(2) if so, insure that <u>all</u> firings in $B_\Omega(e_2)$ are sequenced, say, after <u>all</u> firings in $B_\Omega(e_1)$. I.e., insure that in no firing sequence Ω starting in S does a firing in $B_\Omega(e_1)$ follow a firing in $B_\Omega(e_2)$.

The next paragraph explains a general technique for insuring that entire blocking groups are sequenced (step (2) above). This is followed by a characterization of those groups in an L_S program which must be sequenced (according to step (1) above).

The one thing which all firings in $B_\Omega(e_2)$ have in common on the tagged interpreter is that they remove tokens with tag e_2. None of these firings occurs until after the first appearance (on arcs of the configuration) of such tokens. Therefore, a simple sequencing technique is to insure that the first tokens with tag e_2 do not appear on arcs until all firings in $B_\Omega(e_1)$ have occurred. This implies directly that the first appearance of a token with tag e_1 (which precedes all firings in $B_\Omega(e_1)$) must precede the first appearance of a token with tag e_2. Given that the first tokens with tag e_1 have appeared, an easily-implemented indication that all firings in $B_\Omega(e_1)$ have occurred is the disappearance of the last such token. (This is because for all firings φ in Ω except one, if φ places a token with tag e_1 on an arc, there must have been such a token on another arc for φ to remove.) These two observations give rise to

The Group Sequencing Technique: For any firing sequence Ω, if

1. the first tokens with tag e_1 appear before the first tokens with tag e_2, and

2. the first tokens with tag e_2 do not appear while there are still tokens with tag e_1 in the configuration,

then no firing in $B_\Omega(e_2)$ precedes a firing in $B_\Omega(e_1)$.

This is a general technique for sequencing all firings in one blocking group after all firings in another. Now a rule is developed establishing when two blocking groups in an L_S program should be so sequenced.

Let d_1 and d_2 be any two actors in an L_S program such that, in some firing sequence Ω, a firing φ_1 of d_1 potentially interferes with a firing

φ_2 of d_2, and φ_1 and φ_2 are in distinct blocking groups $B_\Omega(e)$ and $B_\Omega(e')$. Then at least one of the actors, say d_1, must be in the write-class (by Table 3.1-1). Firing φ_1 removes a token with tag e, by definition of blocking group. The following conclusions about e follow from the Static/Dynamic Group Relationship:

1. If $e = Tg(ID,i)$ for some i, then d_1 is in $G(K(ID,i))$.

2. If $e = Tg(S,n)$ for some n, where S is a Select operator, then d_1 is in $G(K(S,1))$.

It has already been argued from Figure 3.2-3 that for each write-class operator d in an L_S program, the primary input arc of d is an output arc of a Copy operator; i.e., d_1 is in $G(K(C,1))$ or $G(K(C,2))$ for some Copy operator C. Therefore, both 1. and 2. above are contravened, so

the firing φ_1 of d_1 removes a token with tag $e = Tg(C,n)$ for some n, where C is a Copy operator.

Let p be the pointer such that tagged pointer $TP(p,Tg(C,n))$ is the value of that token removed by φ_1; then that tagged pointer is the output of the nth firing of C. Since φ_2 is in $B_\Omega(e')$, it removes a token with tag e'; let p' be such that $TP(p',e')$ is the value of that token. Then the following conclusions can be drawn about e':

1. If $e' = Tg(C',j)$ for some j, where C' is a Copy, then p' is the pointer output by the jth firing of C'. Since $e' \neq e$, this is not the nth firing of C. Since each Copy firing in Ω outputs a unique pointer, $p' \neq p$. But then φ_1 and φ_2 do not potentially interfere.

2. If $e' = Tg(ID,i)$ for some i, then there are tagged pointers with pointer p' in the initial state. This implies that p' is in the

initial Π, and hence is unequal to any pointer p output by a Copy

firing in Ω.

Therefore,

the firing φ_2 of d_2 removes a token with tag $e' = Tg(S,j)$ for some

j, where S is a Select operator.

This is as far as the identification problem will be resolved.
That is, the following will be assumed as the answer to the question of
which distinct blocking groups contain potentially-interfering firings:

The Potential-Interference Assumption: Given a firing sequence Ω and two

distinct blocking groups $B_\Omega(e)$ and $B_\Omega(e')$, some firing in one

group potentially interferes with some firing in the other iff:

1. $e = Tg(C,n)$ for some n, where C is a Copy operator,

2. $e' = Tg(S,j)$ for some j, where S is a Select operator, and

3. the jth firing of S outputs the same pointer as the nth firing

of C.

Now the strategy for sequencing all potentially-interfering firings
in distinct blocking groups can be seen: Insure that any two groups
which are assumed by the above to contain potentially-interfering firings
are mutually sequenced by the Group Sequencing Technique; i.e., all
firings in one group are sequenced after all firings in the other. This
strategy is most easily implemented by imposing the following restriction:

The Blocking Discipline: For every Select operator S, integer $j > 0$, and

pointer p, tokens with value $TP(p,Tg(S,j))$ do not appear on the

output arcs of S so long as any arcs hold tokens with value

$TP(p,Tg(C,n))$ where C is a Copy and n is any integer.

The effectiveness of the discipline is readily seen in the next paragraph; an evaluation in terms of ease of implementation and unnecessary sequencing of firings which do not potentially interfere is in Chapter 8.

Let e and e' be any two tags such that, in firing sequence Ω, $B_\Omega(e)$ and $B_\Omega(e')$ should be mutually sequenced, according to the strategy. By the Potential-Interference Assumption, one of the tags, say e', is $Tg(S,j)$ where S is a Select operator, e is $Tg(C,n)$ where C is a Copy, and the jth firing of S outputs the same pointer p as the nth firing of C. Since the node to which p points is activated by the Copy firing, the Select firing could not have output p before that firing. I.e., the output tokens of the Copy firing, which are the first to appear with tag $Tg(C,n)$, appear before the output tokens of the Select firing, which are the first with tag $Tg(S,j)$. By the Group Sequencing Technique, then, $B_\Omega(e)$ and $B_\Omega(e')$ will be mutually sequenced if the output tokens of the Select firing do not appear while there are tokens with tag $Tg(C,n)$ on any arcs. The former tokens appears on S's output arcs, and have value $TP(p,Tg(S,j))$. The latter tokens all have value $TP(p,Tg(C,n))$. Therefore $B_\Omega(e)$ and $B_\Omega(e')$ will be mutually sequenced if no tokens with value $TP(p,Tg(S,j))$ appear on S's output arcs so long as any arc holds a token with value $TP(p,Tg(C,n))$. This will be the case under the Blocking Discipline.

Enforcing the Blocking Discipline requires comparing, at every Select firing, the output produced by that firing against the values of all pointer-valued tokens existing in the configuration. It may be discovered at the jth firing of Select S that its output tokens, which have

value $TP(p,Tg(S,j))$, cannot be placed on S's output arcs immediately
(because a token with value $TP(p,Tg(C,n))$ was found). In this case, the
label S will be placed in a <u>pool</u>, which is separate from the configuration
and heap of the state and is associated with pointer p. S will be removed
from this pool, and tokens of value p placed on the output arcs of the
actor labelled S, after the last tokens with value $TP(p,Tg(C,n))$ disappear
from the configuration. Incorporating an optimized version of this mech-
anism into the standard data-flow interpreter yields the <u>modified</u> <u>data-</u>
<u>flow interpreter</u>, described in the next section. First it is briefly
demonstrated that this mechanism does insure the sequencing of every pair
of potentially-interfering firings in AlterS' (on the tagged interpreter).

It has already been argued that the potentially-interfering firings
of U and S_2, which are always in the same blocking group, are sequenced
on any data-flow interpreter. The only other potentially-interfering
firings in AlterS' are of A and F. In any firing sequence Ω, these
firings are in the distinct blocking groups $B_\Omega(Tg(C_2,1))$ and $B_\Omega(Tg(S_2,1))$,
respectively. Both firings input the pointer p which points to node n_2
(Figure 3.2-1). That pointer is output first by the firing of C_2. That
firing places tokens with value $t = TP(p,Tg(C_2,1))$ on C_2's output arcs,
which enables A, before S_2 fires. If A fires before S_2, then certainly
A fires before F. If S_2 fires before A, then there will still be tokens
with value t on some arcs (A's input arc). By the Blocking Discipline,
the tokens produced by that firing of S_2, which have value $TP(p,Tg(S_2,1))$,
cannot be placed on S_2's output arcs immediately. Instead, the label S_2

will be placed in a pool associated with p, until such time as A's firing
removes the last token with value t. After that, the label S_2 will be
removed from the pool, and tokens with value $TP(p,Tg(S_2,1))$ will be
placed on S_2's output arcs.

Therefore, even if S_2 fires before A, its output tokens will not be
available to enable F until after A fires. Thus, F always fires after A.
So under the Blocking Discipline, all pairs of potentially-interfering
firings in AlterS' are sequenced; i.e., the program is determinate.

The only language-dependent feature which enters into the argument
in support of the Blocking Discipline is what may be termed
The Read-Only Condition: Every write-class operator is in one of the
m.p.d.g.'s $G(K(C,1))$ or $G(K(C,2))$ where C is a Copy operator.
Therefore, in any L_{BS} program P which satisfies the Read-Only Condition,
every two potentially-interfering firings in distinct blocking groups
are sequenced by the Blocking Discipline. If P also satisfies the
Determinacy Condition, then every two potentially-interfering firings in
the same blocking group are sequenced. Therefore, denoting by L_D the
subset of L_{BS} consisting of the programs which satisfy both Conditions,
every program in L_D is determinate under the Blocking Discipline. The
development of the language L_D and the modified interpreter, which are
formally defined next, has met part of the goal of the thesis; the
translation from L_{BV} to L_D given in Section 3.4 satisfies the remainder.

3.3 The Language L_D

Section 3.3.1 precisely defines the modified interpreter and the Read-Only Condition. Section 3.3.2 then gives a definition of blocking group which is valid on any interpreter, and the detailed Determinacy Condition.

3.3.1 The Modified Data-Flow Interpreter

The modified interpreter is basically just the tagged interpreter with the Blocking Discipline imposed. Two optimizations are made, however. These are motivated in the first sub-section below. Following that are full definitions of the state and the state-transition rule of the modified data-flow interpreter.

3.3.1.1 Optimizations

The only information in a tag which is needed to enforce the Blocking Discipline is whether or not the label in the tag is the label of a Copy operator. It is sufficient, then, that all the tagged pointers ever appearing in a configuration be distinguishable into two classes: those which were output by Copy firings and those which were not. Therefore, the first optimization is to replace tagged pointers with read pointers and write pointers:

Definition 3.3-1 A read pointer is an ordered pair (p,R) where p is a pointer. A write pointer is an ordered pair (p,W) where p is a pointer.

$\triangle$

An initial state of the modified interpreter has no write pointers in it,
Select firings always output read pointers, and write pointers are output
only by Copy firings.

Under the Blocking Discipline, for any two tags $e = Tg(C,n)$, where
C is a Copy, and $e' = Tg(S,j)$, where S is a Select, and for any firing
sequence Ω, if the jth firing of S outputs the same pointer as the nth
firing of C, then all firings in $B_\Omega(e')$ are sequenced after all firings in
$B_\Omega(e)$. By the Static/Dynamic Group Relationship, all firings in $B_\Omega(e')$
are of operators in the m.p.d.g. $G(K(S,1))$. By the Read-Only Condition,
all of these firings are of read-class operators. Therefore, none of
them potentially interferes with any read-class firings which may be in
$B_\Omega(e)$. I.e., it is necessary only that the firings in $B_\Omega(e')$ be
sequenced after all the write-class firings in $B_\Omega(e)$; sequencing them
after the read-class firings as well entails an unnecessary loss of
structure concurrency.

This can be corrected by allowing a firing of C to place read pointers
in those channels which lead only to read-class operators. Then the only
firings in $B_\Omega(e)$ guaranteed to have write pointers as inputs are the
write-class firings (by a refined version of the Static/Dynamic Group
Relationship, proven as Lemma 3.3-1). So the disappearance of the last
such write pointer is a signal only that all write-class firings in $B_\Omega(e)$
have occurred. There may still be read-class firings in $B_\Omega(e)$ which have
not occurred; the Blocking Discipline will not sequence these with respect
to any of the firings in $B_\Omega(e')$.

Accordingly, on the modified interpreter, the two groups of output arcs of a Copy operator C will get slightly different tokens: Every firing of C places write pointers (p,W) only on its number-1 output arcs, while placing read pointers (p,R) on its number-2 output arcs. Additionally, the Read-Only Condition is refined:

Definition 3.3-2 An L_{BS} program P satisfies the Read-Only Condition iff for every write-class operator d in P, d is in the m.p.d.g. G(K) only if K = K(C,1) for some Copy operator C.

$\triangle$

3.3.1.2 The Modified State

A state of the modified data-flow interpreter differs from a standard interpreter state in two regards. First is the replacement of simple pointers in the configuration by read and write pointers:

Definition 3.3-3 A modified configuration of a data-flow program P consists of:

1. P, plus

2. an association of

    i. a non-pointer value, or

    ii. a read pointer or a write pointer, or

    iii. the symbol null

    with each data arc of P, plus

3. an association of a symbol from the set {true, false, null} with each control arc of P.

$\triangle$

The second distinctive feature of a modified state is a third component, containing the pools of labels of Select operators whose output tokens are being withheld.

Definition 3.3-4 A <u>modified</u> <u>interpreter</u> <u>state</u> is an ordered triple (Γ, U, Q) where

Γ is a modified configuration

U is a heap (Definition 2.2-1), and

$$Q: V_p \to 2^L$$

is the <u>pool</u> <u>component</u>, which associates set of

actor labels with certain pointers.

$\triangle$

As mentioned, only read pointers appear in an initial state of the modified interpreter:

Definition 3.3-5 A modified state (Γ, U, Q) is an <u>initial</u> <u>modified</u> <u>state</u> for program P iff:

1. there is an initial standard state (Γ', U) for P such that Γ is Γ' with each pointer p which is associated with an arc replaced by the read pointer (p, R), and

2. Q is empty.

$\triangle$

Clearly there is a one-to-one correspondence between initial standard states and initial modified states.

3.3.1.2   The Modified State-Transition Rule

The state-transition rule for the modified interpreter differs from that for the standard interpreter in two relatively minor regards: (1) all pointer inputs and outputs of a structure operator are read and write pointers, and (2) the appearance of the output tokens of a Select firing may be delayed, in accordance with the Blocking Discipline. In addition, the enabling conditions for an actor to fire must be augmented to disallow enabling a Select whose output tokens are being withheld.

Definition 3.3-6   Given a modified interpreter state (Γ, U, Q), any actor d in Γ is enabled (to fire) iff

1.   the distribution of tokens on d's input and output arcs in Γ matches the enabled condition for d, according to Definition 2.1-4, and

2.   if d is a Select operator, there is no pointer p such that $d \in Q(p)$.

$\triangle$

The strong connection between the standard and the modified state-transition rules is made most evident if these are treated as state-transition functions. That is, the standard rule may be considered to define two functions from the current state and an actor enabled in it to the new standard state:

Definition 3.3-7   The standard state-transition functions

$$\text{Standard}_\Gamma((\Gamma, U), d) \quad \text{and} \quad \text{Standard}_U((\Gamma, U), d)$$

are defined for any standard state (Γ, U) and any actor d enabled in Γ. Their values are the configuration and heap, respectively, of the new

state derived by applying the standard state-transition rule to (Γ, U) with d chosen as the actor to fire.

$\triangle$

These functions cannot be used directly for the modified interpreter, because they are not defined when a structure operator's input is a read or a write pointer. This incompatibility is rectified by:

Definition 3.3-8 The function

$$Strip(\Gamma, d)$$

is defined for any modified configuration Γ and actor d in Γ to be identical to Γ, except that if d is a structure operator, each input token of d which has value (p, R) or (p, W) is replaced by a token with value p.

$\triangle$

Now $Standard_{\Gamma}((Strip(\Gamma, d), U), d)$ and $Standard_{U}((Strip(\Gamma, d), U), d)$ are defined for any modifed state (Γ, U, Q) and actor d enabled in Γ.

Obeying the Blocking Discipline optimally requires a two-step state transition. In the first step, an enabled actor d is fired: The appropriate tokens are removed from its input arcs, and, if it is not a Select, the appropriate tokens are placed on its output arcs, exactly as in the standard interpreter (except for the R and W tags in pointers). If d labels a Select, however, then the label d is placed in the pool $Q(p)$, where p is the pointer which this firing would have output on the standard interpreter. The result of applying this first step to modified state (Γ, U, Q) and enabled actor d will be denoted $Fire((\Gamma, U, Q), d)$.

The second step is to release any Select output tokens which can now be allowed to appear on arcs of the configuration. The decision on whether to place tokens of value (p,R) on any arcs is based on the presence or absence of any tokens with value (p,W). If there are none, then tokens with value (p,R) are placed on all the data output arcs of the Selects labelled by all the labels in the pool Q(p). The result of applying this second step to any modified state (Γ,U,Q) will be denoted Release((Γ,U,Q)). Therefore, the overall state-transition function for the modified inter-preter is Release(Fire($(\Gamma,U,Q),d$)). The reason for a two-step transition is a subtle one, and will be given after the following precise statement of the rule:

Definition 3.3-9 The state-transition rule for the modified data-flow interpreter is:

Given a state (Γ,U,Q) in a state sequence, each possible next state in the sequence is found by:

1. Choose one actor d enabled to fire in Γ.

2. The next state is then Release(Fire($(\Gamma,U,Q),d$)), where the functions Fire and Release are defined below.

Let Γ_S=Standard$_\Gamma$((Strip(Γ,d),U),d) and U_S=Standard$_U$((Strip(Γ,d),U),d).

Fire($(\Gamma,U,Q),d$) is defined by:

1. If d is not a Copy or a Select operator, then

$$Fire((\Gamma,U,Q),d) = (\Gamma_S,U_S,Q)$$

2. If d is a Copy operator, let pointer p be the value of the tokens on d's data output arcs in Γ_S. Let Γ' be Γ_S with the tokens on d's

number-1 output arcs having value (p,W) and the tokens on d's

number-2 output arcs having value (p,R). Then

$$Fire((\Gamma,U,Q),d) = (\Gamma',U_S,Q)$$

3. If d is a Select operator:

a. If the value of the tokens on d's output arcs in Γ_S is <u>undef</u>, then

$$Fire((\Gamma,U,Q),d) = (\Gamma_S,U_S,Q)$$

b. Otherwise, let pointer p be the value of the tokens on d's data

output arcs in Γ_S. Let Γ' be Γ_S with these tokens removed, and

let Q' denote the function

$$Q'(r) = \begin{cases} Q(r) & \text{if } r \neq p \\ Q(p)\cup\{d\} & \text{if } r = p \end{cases}$$

Then

$$Fire((\Gamma,U,Q),d) = (\Gamma',U_S,Q')$$

$Release((\Gamma,U,Q)) = (\Gamma'',U,Q'')$ where Γ'' and Q'' are identical to Γ and Q

except that:

For any pointer p such that Q(p) is non-empty and there are no tokens

with value (p,W) in Γ,

for all $c \in Q(p)$,

Γ'' has tokens of value (p,R) on all of c's data output arcs,

and $Q''(p)$ is the empty set.

$\triangle$

Under this rule, no tokens with value (p,R) appear on the output arcs of a

Select operator except as they are released during the second step of some

state transition. Since this never occurs while there are any tokens with

value (p,W), the Blocking Discipline is obeyed.

It will be noted that the decision to release tokens at the second step of a transition is based on the configuration after the first step. As will be seen in Section 8.2, this two-step transition is easier to implement than a single-step transition. The only semantic significance arises in the case that a Select firing φ inputs the value (p,W) and outputs the value (p,R). (This implies that there is a branch from the node pointed to by p to itself; this is valid in the heaps defined.) If the token removed by φ is the only one with value (p,W), then the output tokens of φ will be released at the second step of the same transition. If the transition were made in one step, however, the decision to release the tokens could be based only on the configuration before the entire transition. Then there would be no choice but to withhold them until the following transition. Thus the two-step transition is easier to implement and may give rise to increased concurrency.

This completes the formal specification of the modified interpreter embodying the Blocking Discipline. It is proven in the next four chapters that any L_{BS} program satisfying both the Determinacy Condition and the Read-Only Condition is determinate on the modified interpreter. The Read-Only Condition has already been stated precisely. Now the Determinacy Condition and the concept of blocking groups are defined for programs run on the modified interpreter.

3.3.2 The Determinacy Condition

Blocking groups were introduced on the tagged interpreter. Each group was associated with a tag uniquely denoting a program input arc or a

Copy or Select firing. On that interpreter, each pointer-valued token removed by a firing had one of these tags, and the firing was in the blocking group for that tag. But these cumbersome, if convenient, tags have been eliminated from the modified interpreter. Therefore, the concept of tag-bearing tokens is abstracted away from the concept of blocking group in the following definition. At the same time, blocking groups are sub-divided to reflect the slight difference between the number-1 and number-2 outputs of a Copy on the modified interpreter; the utility of this will be seen shortly.

Definition 3.3-10 (Blocking groups) For any firing sequence Ω, starting in any initial state for any program P, denote by $PRF(\Omega)$ the set of firings φ in Ω satisfying one of the following:

a. φ is a firing of a structure operator in P, or

b. φ is a firing of a pI actor in P from a transmitted-input arc of which φ removes a read or a write pointer.

The set of sub-blocking groups in Ω is a partition of $PRF(\Omega)$. The particular sub-blocking group containing any given firing φ of an actor d is determined from the primary input arc b of d as follows:

1. If b is the ith program input arc of P, then φ is in just the sub-blocking group denoted by $SB_\Omega(ID,i)$.

2. If b is a data arc in the number-i group of output arcs of a Copy or Select operator d', then φ is in just $SB_\Omega(d',n,i)$, n being such that the token removed from b by φ is the nth to appear on b in Ω.

3. If b is an output arc of a pI actor d', then φ is in the same sub-blocking group(s) as the firing of d' which placed on b the token removed by φ.

Finally, the <u>blocking groups</u> in Ω are given by:

1. For all i, the blocking group denoted by $B_\Omega(ID,i)$ is just $SB_\Omega(ID,i)$.

2. For any Select operator S and for all n, the blocking group denoted by $B_\Omega(S,n)$ is just $SB_\Omega(S,n,1)$.

3. For any Copy operator C and for all n, the blocking group denoted by $B_\Omega(C,n)$ is $SB_\Omega(C,n,1) \cup SB_\Omega(C,n,2)$.

$\triangle$

With the substitution of "tagged pointer" for "read or write pointer", this definition gives the same set of blocking groups for a firing sequence on the tagged interpreter as did the earlier, informal one.

Now the Determinacy Condition can be made precise:

<u>Definition 3.3-11</u> A program P satisfies the <u>Determinacy Condition</u> iff the following is true for every pair of distinct structure operators d_1 and d_2 in P and every initial state S for P: Let Ω be any firing sequence starting in S in which the ith firing of d_1 and the jth firing of d_2 are in a common blocking group and potentially interfere. For any other firing sequence Ω' starting in any state equal to S, the ith firing of d_1 and the jth firing of d_2 appear in the same relative order in Ω' as in Ω.

$\triangle$

With this, the language L_D is fully specified:

Definition 3.3-12 The <u>determinate</u> <u>structure-as-storage</u> <u>data-flow</u> <u>language</u>, L_D, consists of those L_{BS} programs which satisfy both the Determinacy Condition and the Read-Only Condition.

$\triangle$

A syntactic characterization of all L_{BS} programs which satisfy the Determinacy Condition has yet to be found. It is known that the complexity of any such characterization is reduced by the fact that if firings of two actors are in a common sub-blocking group, then the actors are in the same m.p.d.g. This follows from the Static/Dynamic Group Relationship, a comprehensive version of which can now be proven formally:

Definition 3.3-13 An L_{BS} program P satisfies the <u>Static/Dynamic</u> <u>Group</u> <u>Relationship</u> iff for every firing sequence Ω starting in any initial state S for P, A and B below are true for every firing φ in $PRF(\Omega)$. Let d be the actor of which φ is a firing, and let v be the value removed by φ from d's primary input arc.

  A: Exactly one of the following two statements is true of φ:

  1. There is exactly one integer i such that φ is in $SB_\Omega(ID,i)$, d is in $G(K(ID,i))$, and there is a token of value v on P's number-i program input arc in S.

  2. There is exactly one Copy or Select operator S in P, and exactly one integer n and one integer i, such that φ is in $SB_\Omega(S,n,i)$, d is in $G(K(S,i))$, the nth tokens to appear on S's number-i group of output arcs in Ω have value v, and that appearance does not follow the appearance of the token removed by φ.

B: Value v is a write pointer iff φ is in $SB_\Omega(C,n,1)$ for some Copy operator C and some integer n>0.

$\triangle$

Lemma 3.3-1 Every L_{BS} program running on the modified interpreter satisfies the Static/Dynamic Group Relationship.

Proof: Let S be any initial modified state of any L_{BS} program P. Proof is by induction on the length of the firing sequences starting in S.

Induction hypothesis is that A and B are true for every firing in every length-n firing sequence starting in S.

Basis: n = 0. Vacuously true.

Induction step: Assume the induction hypothesis is true for n = k≥0, and consider it for n = k+1.

(1) Let $\Omega = \theta\varphi$ be any length-n firing sequence starting in S. Then A and B are true for every firing in θ ind. hyp.

(2) If φ is not in PRF(Ω), then A and B are true for every firing in Ω

                                                                           (1)

(3) Assume φ is in PRF(Ω). Let d be the actor of which φ is a firing, let b be d's primary input arc, and let v be the value of the token removed from b by φ. Then v is a read or a write pointer

                                    Defs. 3.3-10+3.2-1+2.2-5

(4) The token removed from b by φ either was on b in S or was placed there as the result of a state transition of the modified interpreter. That token is on b in $S \Rightarrow$ b is a program input arc

                                    (3)+Defs. 3.3-5+2.2-6

(5) That token was placed on b at a transition $\Rightarrow$ b is a data output arc

of a Copy or Select operator, or a token of pointer value can be placed on b at a transition of the standard interpreter

$$(3)+\text{Defs. } 3.3\text{-}9+3.3\text{-}7$$

(6) $\Rightarrow$ b is a data output arc of a Copy, Select, or pI actor   Def. 2.2-4

(7) b is either a program input arc or a data output arc of a Copy, Select, or pI operator   (4)+(5)+(6)

Case I: b is the number-i program input arc

(8) φ is in just $SB_\Omega(ID,i)$, and so is not in $SB_\Omega(C,n,1)$ for any Copy operator C and integer n   Def. 3.3-10

(9) b is in $K(ID,i)$, so b is in a channel starting at $b \in K(ID,i)$, so d is in $G(K(ID,i))$ Def. 3.2-1

(10) b is not an output arc of any actor   Def. 2.1-1

(11) No state transition can cause a token to be placed on b, so the token removed from b by φ, which is of value v, is on b in S

$$(10)+\text{Defs. } 3.3\text{-}9+2.2\text{-}5+2.1\text{-}5$$

(12) v is a read pointer   (3)+Def. 3.3-5

Case II: b is a data output arc of a Copy or Select operator S

(13) There is exactly one n such that the token removed from b by φ is the nth to appear there in Ω, and there is exactly one i such that b is in the number-i group of output arcs of S   Def. 2.1-1

(14) φ is in just $SB_\Omega(S,n,i)$ (13)+Def. 3.3-10

(15) b is in $K(S,i)$, so d is in $G(K(S,i))$ Def. 3.2-1

(16) The nth set of tokens to appear on the number-i group of output arcs of S in Ω includes the token removed from b by φ, and so those tokens have value v and their appearance does not follow that of the token removed by φ (13)

(17) v is a write pointer iff S is a Copy and i=1   (13)+Def. 3.3-9

(18) iff $\varphi \in SB_\Omega(C,n,1)$ for some Copy C and integer n   (14)

Case III: b is an output arc of a pI actor d'

(19) There is a prefix $\Delta\varphi'$ of θ such that the token removed from b by φ

is not on b in $S \cdot \Delta$, but is on b in $S \cdot \Delta\varphi'$, so φ' is a firing of d'

and, letting $S \cdot \Delta$ be (Γ,U,Q), there is a token of value v on b in

$\text{Standard}_\Gamma((\text{Strip}(\Gamma,d'),U),d')$ Def. 3.3-9

(20) φ' removes a token of value v which is on a transmitted-input arc

a of d' in $\text{Strip}(\Gamma,d')$ (19)+Defs. 3.3-7+2.2-4

(21) φ' removes a token of value v from a primary input arc a of d'

which is on a in Γ (20)+Defs. 3.2-1+3.3-8

(22) φ is in the same sub-blocking group(s) as φ' (19)+Def. 3.3-10

(23) b is in every channel a is in, so d is in every distribution group

d' is in   Def. 3.2-1

(24) φ' is in θ, so it is in $\text{PRF}(\theta)$ (21)+Def. 3.3-10

(25) Either φ' is in $SB_\theta(ID,i)$ for exactly one i, or φ' is in $SB_\theta(S,n,i)$

for exactly one Copy or Select operator S, one n, and one i

(24)+(1)+Def. 3.3-10

(26) Either φ' is in $SB_\Omega(ID,i)$ for exactly one i, or φ' is in $SB_\Omega(S,n,i)$

for exactly one Copy or Select operator S, one n, and one i

(25)+Def. 3.3-10

(27) Either φ is in $SB_\Omega(ID,i)$ for exactly one i, or φ is in $SB_\Omega(S,n,i)$

for exactly one Copy or Select operator S, one n, and one i(26)+(22)

(28) For any i, $\varphi \in SB_\Omega(ID,i) \Rightarrow \varphi' \in SB_\Omega(ID,i) \Rightarrow [d' \in G(K(ID,i))$ and there is

a token of value v on P's number-i program input arc in $S] \Rightarrow$

$d \in G(K(ID,i))$ (22)+(24)+(1)+(23)+(21)+Def. 3.3-13

(29) For any Copy or Select operator S, any n, and any i, $\varphi \in SB_\Omega(S,n,i)$

$\Rightarrow \varphi' \in SB_\Omega(S,n,i) \Rightarrow d' \in G(K(S,i))$ and the nth tokens to appear on

S's number-i output arcs in θ have value v, and that appearance

does not follow the appearance of the token removed from a by φ'

(22)+(24)+(1)+(21)+Def. 3.3-10

(30) The appearance of the token removed from b by φ follows the

appearance of the token removed from a by φ' (21)+(19)

(31) $\varphi \in SB_\Omega(S,n,i) \Rightarrow d \in G(K(S,i))$ and the appearance of the nth set of

tokens to appear on S's number-i group of output arcs does not fol-

low the appearance of the token removed from b by φ (29)+(23)+(30)

(32) v is a write pointer iff $\varphi' \in SB_\Omega(C,n,1)$ for some Copy operator C

and integer n > 0   (21)+(24)+(1)+Def. 3.3-13

(32) iff $\varphi \in SB_\Omega(C,n,1)$ (22)

$\triangle$

    The Determinacy Condition concerns only pairs of structure operator

firings in a common blocking group. Therefore, any syntactic test for

this Condition need consider only pairs of structure operators in the same

or in closely-related m.p.d.g.'s. Specificially, any pair of such opera-

tors d_1 and d_2 must be in either $G(K(ID,i))$ for some i, $G(K(S,1))$ for

some Select operator S, or $G(K(C,1)) \cup G(K(C,2))$ for some Copy operator C.

Table 3.1-1 may reveal that no firing of d_1 potentially interferes with

any firing of d_2. Otherwise, one of d_1 and d_2 must be in the write class.

If the Read-Only Condition is satisfied (which is easily confirmed), then

that write-class operator must be in $G(K(C,1))$ for some Copy operator C.

Thus both d_1 and d_2 must be in $G(K(C,1)) \cup G(K(C,2))$, and some firings of

them may have to be sequenced.

The only syntactic test for this sequencing which is known to be valid covers an important special case: If none of the channels starting at arcs in $K(C,1) \cup K(C,2)$ contains an input or output arc of a gate, then it is sufficient that there is a directed path from d_1 to d_2 which is similarly free of gate input and output arcs. Furthermore, the following, quite general test, is believed to be correct: A <u>well-formed</u> data-flow program is an acyclic interconnection of individual actors, conditional constructs like Figure 2.1-6, and iteration constructs in the fashion of Figure 2.1-4 [13]. In a well-formed program, it is sufficient that d_1 and d_2 either:

1. are in separate subprograms of a conditional construct, or

2. have a directed path between them.

Finally, it is now possible to fully appreciate the decisions to associate two m.p.d.g.'s with each Copy operator and two sub-blocking groups with each Copy firing. Both help simplify the proof that, in a program satisfying the Read-Only Condition, all firings of write-class operators input write pointers; this in turn is a key to the effectiveness of the Blocking Discipline. The proof consists of two simple steps:

1. For every write-class operator d, there is a Copy operator C such that d is in $G(K(C,1))$ (Read-Only Condition).

2. Every firing of d is in $SB_\Omega(C,n,1)$ for some n, and every firing of d inputs a write pointer (Static/Dynamic Group Relationship).

Clearly, without the separate notations for $G(K(C,1))$ and $SB_\Omega(C,n,1)$, the expression of this proof would be considerably less elegant.

3.4 The Translation

The previous section has specified the data-flow language L_D and the modified data-flow interpreter. It is proven in succeeding chapters that on the modified interpreter, any L_D program is determinate, hence functional. This partially meets the primary goal of the thesis; the remaining requirement is satisfied in this section, by presenting an algorithm which translates any well-behaved L_{BV} program into an equivalent L_D program.

The translation is an improved version of the simplistic one given earlier (Algorithm 3.2-1). That algorithm replaces each Const, Append, and Remove operator in an L_{BV} program with a combination of L_{BS} operators: a Copy C, a sequencer G, and an Assign, Update, or Delete U, arranged as in Figure 3.2-3. A minor refinement is obviously needed to guarantee that the resulting L_{BS} program satisfies the Read-Only Condition: U's primary input arc is made the only number-1 output arc of the Copy. In this way, the single write-class firing in any blocking group is the only one to input a write pointer.

Every firing of U is always sequenced before every other firing in the same blocking group. This is because, for every actor $d \neq U$ of which there is a firing in that blocking group, d is in the m.p.d.g. $G(K(C,2))$; i.e., its primary input arc is in a channel starting at an output arc of C, and every such channel goes through G. There may be, however, an output arc b of C such that all channels starting at b end at number-3 input arcs of other Updates. None of those other Updates is in $G(K(C,2))$, so none of their firings is in the same blocking group as any firing of U. Therefore, having all those channels go through G results in an unnecessary

loss of structure concurrency. Easing this constraint may in turn lead to G's having no output arcs, in which case G itself, along with its input arcs, can be removed. These considerations are precisely stated in the following.

<u>Algorithm 3.4-1</u> This algorithm constructs, from any L_{BV} program P, an L_{BS} program P'. It also constructs two maps: T, from the actor labels in P to those in P', and A, from the arcs in P to those in P'.

Let L_P be the set of labels of the actors in P. Let T be any function

$$T: L_P \rightarrow L_P \cup (L - L_P)^3$$

such that:

1. If d does not label a Const, Append, or Remove, then $T(d) = d$.

2. Otherwise, $T(d)$ is a triple (C,U,G) of labels not in L_P.

3. No label appears more than once in all of the triples in the range of T.

Then P' is the unique L_{BS} program, and A the unique map, satisfying the following specifications:

For each actor in P, let d be its label. Then:

The actor is not a Const, Append, or Remove iff there is an actor of the same type in P', labelled with $T(d)$.

The actor is a Const/Append/Remove iff there are three actors in P' labelled with the labels in the triple $T(d) = (C,U,G)$, and C labels a Copy, G a sequencer, and U an Assign/Update/Delete.

For each arc b in P:

If b is not an input or an output arc of a Const, Append, or Remove, then b is an input (output) arc of the actor labelled d in P iff

A(b) is the same input (output) arc of the actor labelled T(d) in P'.

If b is an input or output arc of a Const, Append, or Remove labelled

d in P, let T(d) = (C,U,G). Then

1. b is the number-1 input arc of d iff A(b) is the number-1
    input arc of C (Figure 3.4-1).

2. b is the number-2 (number-3) input arc of d iff A(b) is the
    number-2 (number-3) input arc of U.

3. b is a control output arc of d iff A(b) is a control output
    arc of U.

4a. If b is a data output arc of d and every channel in P starting
    at b ends at a number-3 input arc of an Append, then A(b) is a
    number-2 output arc of C.

4b. b is a data output arc of d and not every channel in P starting
    at b ends at a number-3 input arc of an Append iff A(b) is an
    output arc of G.

Finally, for each Const, Append, or Remove actor d in P, let T(d) be

(C,U,G). Then there are three arcs in P' interconnecting C, U, and G as

in Figure 3.4-1 (these arcs are not in the map A). The input arc of U is

the only arc in the number-1 group of output arcs of C. (If sequencer G

has no output arcs, it may be removed, along with its input arcs.)

$\triangle$

Clearly A as defined is a similarity mapping from P to P'.

Figure 3.4-2 depicts the L_{BS} program AlterS2', which is the result of

translating the L_{BV} program AlterV2 (Figure 2.3-7a). Note that the only

difference between AlterS2' and AlterS2 (Figure 2.3-7b) is the insertion

of a sequencer which forces S_3 to fire after U_1.
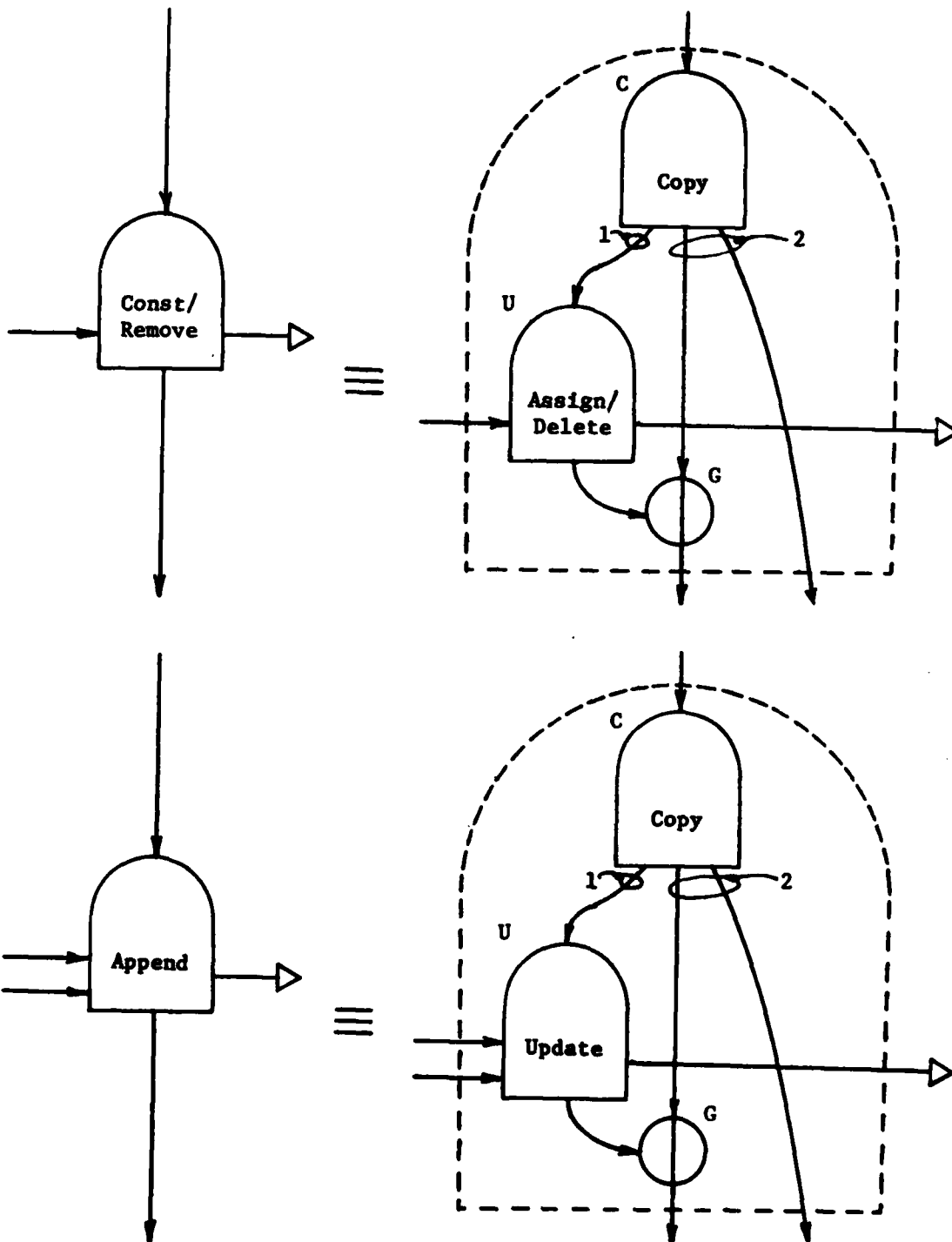
Operator Substitutions in Translating from L_{BV} to L_D

Figure 3.4-1
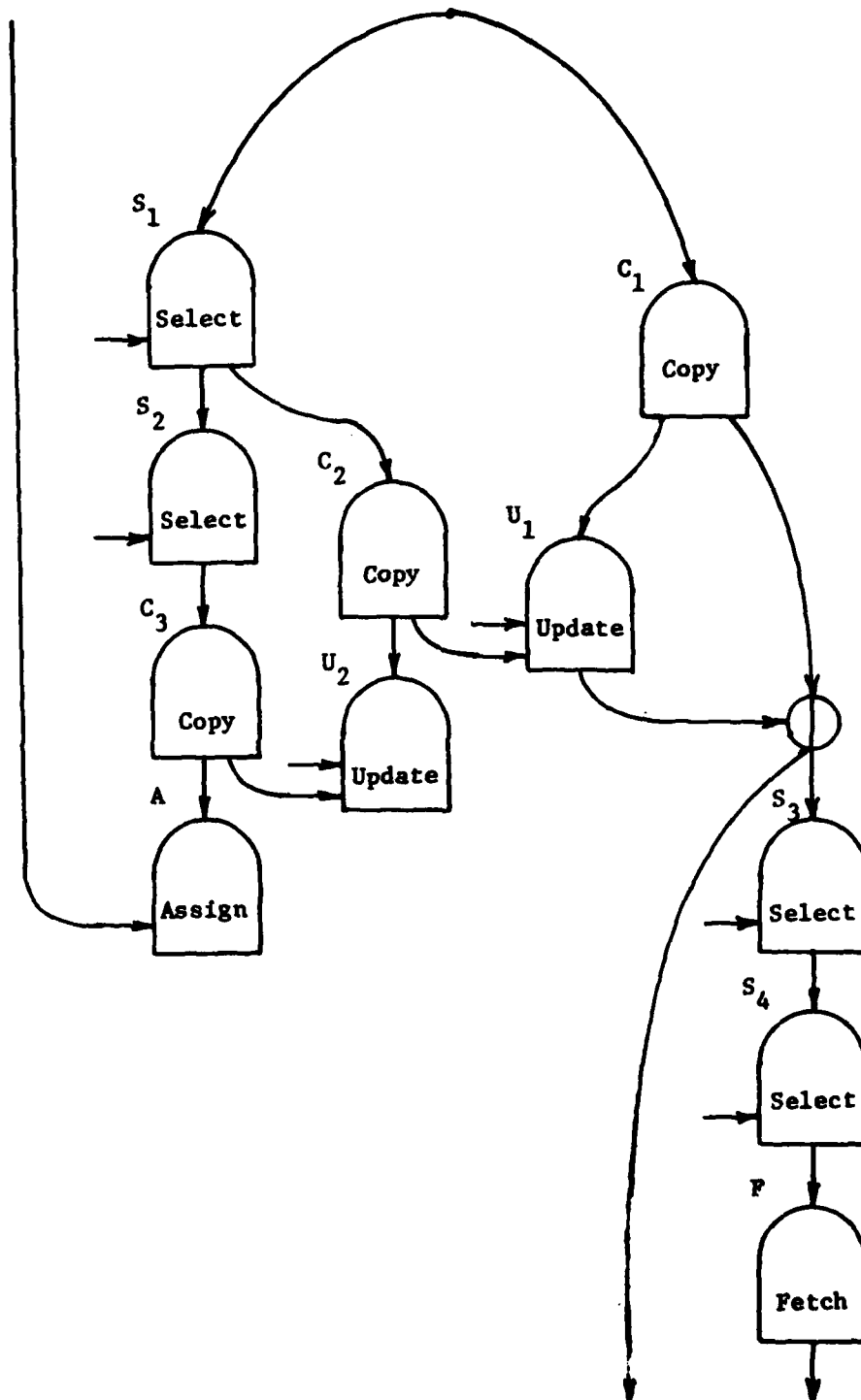
The Program AlterS2'

Figure 3.4-2

The proof that this algorithm translates any well-behaved L_{BV} program P into an equivalent program P' is in three steps:

1. P' satisfies the Read-Only and Determinacy Conditions (i.e., $P' \in L_D$).

2. Every L_D program is functional.

3. If P is well-behaved and P' is functional, then P' is equivalent to P.

The first and third steps are presented below. The proof of the second step occupies the next four chapters.

Theorem 3.4-1 Let P' be any L_{BS} program produced by Algorithm 3.4-1 as the translation of some L_{BV} program P. Then P' satisfies the Read-Only Condition and the Determinacy Condition.

Proof:

(1) Let U be any write-class operator in P'

(2) U is not in P   (1)+Defs. 3.1-2+2.2-3

(3) U is introduced into P' by Algorithm 3.4-1   (1)+(2)

(4) There is a unique Copy C in P' connected to U as in Figure 3.4-1

                                                    (1)+(3)+Alg. 3.4-1

(5) The primary input arc b of U is in the number-1 group of output

    arcs of C   (4)+Alg. 3.4-1+Def. 3.2-1

(6) There is exactly one channel containing b which starts at a program

    input arc of P' or a data output arc of a Select or Copy operator,

    and that starts at a number-1 output arc of C   (5)+Def. 3.2-1

(7) The only m.p.d.g. containing U is G(K(C,1))   (5)+(6)+Def. 3.2-1

(8) P' satisfies the Read-Only Condition   (7)+Def. 3.3-2

(9) Let Ω be any firing sequence starting in any initial state S for P'

(10) Let d_1 and d_2 be any two structure operators in P' such that two

distinct firings φ_1 of d_1 and φ_2 of d_2 are both in the same

blocking group in Ω and they potentially interfere

(11) One of d_1 and d_2 is write-class; let it be d_1 (10)+Table 3.1-1

(12) The primary input arc of d_1 is the only number-1 output arc of

some Copy operator C   (11)+(1)+(4)+(5)

(13) The nth firing of any non-gate actor d removes from each input arc

of d the nth token to appear there, and places on each output arc

of d the nth token to appear there   Def. 2.1-5

(14) Let n be such that φ_1 is the nth firing of d_1 in Ω. Then φ_1

removes the nth token to appear on its primary input arc   (13)

(15) $\varphi_1 \in B_\Omega(C,n)$, and no other firing of d_1 is in $B_\Omega(C,n)$

(12)+(14)+Def. 3.3-10

(16) $\varphi_2 \in B_\Omega(C,n)$ (15)+(10)

(17) φ_2 is in $SB_\Omega(C,n,1)$ or $SB_\Omega(C,n,2)$ (16)+Def. 3.3-10

(18) d_2 is in $G(K(C,1))$ or $G(K(C,2))$ (17)+Lemma 3.3-1+Def. 3.3-13

(19) The only channel starting at a number-1 output arc of C contains

just the number-1 input arc of d_1 (12)+Alg. 3.4-1+Def. 3.2-1

(20) d_1 is the only actor in $G(K(C,1))$ (19)+Def. 3.2-1

(21) d_2 is in $G(K(C,2))$ (18)+(20)+(10)

(22) There is a label R of an actor in P and a label G of a sequencer

in P' such that $T(R) = (C,d_1,G)$ (11)+(12)+Alg. 3.4-1

(23) The primary input arc of G is an output arc of C, and its other

input arc is the output arc of d_1 (22)+Fig. 3.4-1

(24) The nth firing of G, φ_G, removes the nth tokens to appear on both

of its input arcs (13)

(25) φ_G is in $B_\Omega(C,n)$, and no other firing of G is in $B_\Omega(C,n)$

(23)+(24)+Def. 3.3-10

(26) φ_G follows φ_1 in Ω (14)+(24)+(13)

Next prove the following, by induction on the length of Ω:

A: Let d be any actor such that d$\in$G(K(C,2)) but d$\neq$G. Then any firing

of d which is in $B_\Omega(C,n)$ follows φ_G in Ω.

Basis: $|\Omega| = 0$. Vacuously true.

Induction step: Assume A is true for every firing sequence of length n,

and consider $\theta\varphi$ of length n+1. If φ is not in $B_\Omega(C,n)$, then A is true

for $\theta\varphi$ by induction hypothesis. Therefore, assume

(27) $\varphi\in B_\Omega(C,n)$ and is a firing of d$\in$G(K(C,2))

(28) There is at least one channel starting at an arc in the number-2

group of output arcs of C and containing the primary input arc

of d (27)+Def. 3.2-1

(29) Since the number-3 input arc of an Update is not its primary input

arc, that channel must start at G's transmitted-input arc

(28)+Def. 3.2-1+Alg. 3.4-1

(30) Since G is a pI actor, that channel also includes an output arc

of G (29)+Def. 3.2-1

(31) Since d$\neq$G, no primary input arc of d is an output arc of C

(29)+Def. 3.2-1

(32) The primary input arc b of d from which φ removes a token is an

output arc of a pI actor d', and that token was placed on b by a

firing φ' of d' also in $B_\Omega(C,n)$ (31)+Def. 3.3-10

(33) φ follows φ' (32)+Def. 2.1-5

There are two cases for d': either d' = G or d' ≠ G.

Case I: d' = G.

(34) φ_G is the only firing of G = d' in $B_\Omega(C,n)$ (25)

(35) $\varphi_G = \varphi'$ (32)+(34)

(36) φ follows φ_G (33)+(35)

Case II: d' ≠ G

(37) $d' \notin G(K(C,2)) \Rightarrow$ there is no channel starting in the number-2 group of

     output arcs of C including a primary input arc of d'   Def. 3.2-1

(38) ⇒ there is no channel starting in the number-2 group of output arcs

     of C which includes b   (32)+Def. 3.2-1

(39) $d' \in G(K(C,2))$ (38)+(27)

(40) φ' is in θ (33)

(41) φ' follows φ_G (39)+(40)+ind. hyp.

(42) φ follows φ_G (41)+(33)

(43) A for $\theta\varphi$ (36)+(42)

(44) $d_2 \in G(K(C,2))$ and $d_2 \neq G$ (21)+(10)

(45) Any firing of d_2 which is in $B_\Omega(C,n)$ follows φ_1 (44)+A+(26)

(46) φ_2 follows φ_1 in Ω (10)+(16)+(45)

(47) If j is such that φ_2 is the jth firing of d_2, then for any other

     firing sequence starting in any initial state for P, the jth

     firing of d_2 follows the nth firing of d_1 (9)+(14)+(46)+(45)

(48) P' satisfies the Determinacy Condition   (9)+(47)+Def. 3.3-11

$\triangle$

     This completes the first step in the proof that Algorithm 3.4-1

translates an L_{BV} program P into an equivalent program P': P' is in L_D.

The third step is presented next: If P is well-behaved and P' is

functional, then P' is equivalent to P. P' is equivalent to P iff for every initial state S for P and halted firing sequence Ω starting in S, for every initial state S' for P' simulating S and halted firing sequence Ω' starting in S', the final state $S' \cdot \Omega'$ simulates $S \cdot \Omega$.

Since P is an L_{BV} program, it is functional; i.e., all halted firing sequences starting in S result in equal firing states. If P' is also functional, then all halted firing sequences starting in S' result in equal final states. The "simulates" relation is invariant under substitution of equal states. Therefore, it is only necessary to find one halted firing sequence Ω starting in S and one halted sequence Ω' starting in S' such that $S' \cdot \Omega'$ simulates $S \cdot \Omega$. The following algorithm constructs such an Ω' from any Ω.

<u>Algorithm 3.4-2</u> Let P be any L_{BV} program, and let T and P' be any map and corresponding L_{BS} program produced from P by Algorithm 3.4-1. Given any firing sequence Ω starting in any initial state for P, and any initial state S' for P', construct a firing sequence $R(\Omega)$ recursively as follows:

Basis: $R(\lambda) = \lambda$.

Induction step: Let $\Omega\varphi$ be any firing sequence starting in any initial state for P, where the last firing φ is of actor d in P. Then $R(\Omega\varphi)$ is given by:

    If d does not label a Const, Append, or Remove, then

        $R(\Omega\varphi) = R(\Omega)\varphi'$, where φ' is the firing which is the label T(d).

    Otherwise, $R(\Omega\varphi) = R(\Omega)\varphi_C\varphi_U\varphi_G$, where

        $T(d) = (C,U,G)$

        $\varphi_C = (C,(p,n))$ where $\varphi = (d,(p,n))$, $\varphi_U = U$, and $\varphi_G = G$. $\triangle$

It should be noted that, technically, the definition of "simulates" (Definition 2.4-7) cannot be applied to a modified state. This is because the definition of the "Match" relation between the conditions of arcs in states (Definition 2.4-2) assumes that each arc has either no token, a token with a non-pointer value, or a token whose value is a simple pointer. In a modified state, however, each token's value is either a non-pointer, or (p,R) or (p,W) where p is a pointer. The association of the innocuous "R" or "W" tags with each pointer in a configuration should not disqualify an otherwise-equivalent program. I.e., it should not be cause for concern if the definition of "simulates" ignores the presence or absence of such tags. This is most easily accomplished by revamping the definition of "Match" (matching conditions for arcs in two modified interpreter states are also included here, for completeness):

Definition 3.4-1 Let S_1 and S_2 be a standard and a modified or two modified interpreter states. Let Γ_1 and Γ_2 be their respective configuration components, and let $U_1 = (N_1, \Pi_1, SM_1)$ and $U_2 = (N_2, \Pi_2, SM_2)$ be their respective heap components. Let b_1 and b_2 each be an arc from the program of which Γ_1 and Γ_2, respectively, is a configuration. Then for any one-to-one mapping $I: N_1 \rightarrow N_2$, the condition of b_2 in S_2 matches under I the condition of b_1 in S_1, written

$$\text{Match}((b_2, S_2), \; I, \; (b_1, S_1))$$

iff one of the following is true:

1. There is no token on b_1 in Γ_1 and no token on b_2 in Γ_2.

2. There are tokens with equal non-pointer values on b_1 in Γ_1 and on b_2 in Γ_2.

3. a. For i=1,2, there is a token with value p_i, (p_i,R), or (p_i,W),

where p_i is a pointer, on b_i in Γ_i,

b. if both tokens' values are tagged pointers, the tags are the
same, and

c. $U_2 \cdot \Pi_2(p_2) \overset{I}{=} U_1 \cdot \Pi_1(p_1)$

$\triangle$

Theorem 3.4-2 Let P be any well-behaved L_{BV} program, and let P' be its
translation via Algorithm 3.4-1. Let S be any initial standard state for
P, and let S' be any initial modified state for P' which simulates S.
Then for any halted firing sequence Ω starting in S:

1. $R(\Omega)$ is a halted firing sequence starting in S', and

2. $S' \cdot R(\Omega)$ simulates $S \cdot \Omega$.

Proof: The proof of this is tedious and has been relegated to Appendix B.
The only non-straightforward points in it are listed below:

1. Whenever a Const, Append, or Remove d is enabled in P, the corres-
ponding Copy C is enabled (Figure 3.4-1). Firing that enables the
corresponding Assign, Update, or Delete U. Firing U then enables
the sequencer G.

2. There is only one write pointer output per Copy firing, and that
is input by the immediately-following firing. Therefore, whenever
a Select fires, there are no write pointers in the configuration,
so its output tokens appear with no delay.

Beyond this, the proof is simply a case-by-case demonstration that equal
inputs (identical non-pointer values or pointers to equal components)
to two firings of an operator produce equal results.

$\triangle$

<u>Theorem 3.4-3</u> For any well-behaved L_{BV} program P, let P' be the L_{BS} program produced from P by Algorithm 3.4-1. If every L_D program is functional, then P' is equivalent to P.

<u>Proof</u>: Let A be the map from arcs in P to arcs in P' generated in the production of P'.

(1) A is a similarity mapping from P to P'   Alg. 3.4-1+Def. 2.4-6

Let S be any initial standard state for P, and let Ω be any halted firing sequence starting in S. Let S' be any initial modified state for P' which simulates S, and let Ω' be any halted firing sequence starting in S'.

(2) $R(\Omega)$ is a halted firing sequence starting in S' and $S' \cdot R(\Omega)$

     simulates $S \cdot \Omega$ Thm. 3.4-2

(3) There is a mapping I_1 such that, for each arc b in P,

     Match$((A(b), S' \cdot R(\Omega)),$ $I_1,$ $(b, S \cdot \Omega))$ (2)+Def. 2.4-7

(4) P' is in L_D Thm. 3.4-1

(5) Every program in L_D is functional $\Rightarrow$ P' is functional   (4)

(6) $\Rightarrow S' \cdot \Omega'$ equals $S' \cdot R(\Omega)$ (2)+Def. 2.4-4

(7) $\Rightarrow$ there is a mapping I_2 such that, for each arc c in P',

     Match$((c, S' \cdot \Omega'),$ $I_2,$ $(c, S' \cdot R(\Omega)))$ Def. 2.4-3

(8) $\Rightarrow$ for each arc b in P,

     Match$((A(b),$ $S' \cdot \Omega'),$ $I_2 \cdot I_1,$ $(b, S \cdot \Omega))$ (3)+Thm. 3.4-1

(9) $\Rightarrow S' \cdot \Omega'$ simulates $S \cdot \Omega$ (1)+Def. 2.4-7

                                                     Q.E.D.

The primary goal of this thesis is to develop a language L_D and an interpreter for it, together with a translation from L_{BV} to L_D which produces equivalent programs having maximal structure concurrency.

Section 3.3 has presented the language L_D and the modified data-flow interpreter for it. It has been argued in Section 3.2 that every L_D program on the modified interpreter is determinate, hence functional. Section 3.4 contains a translation algorithm from L_{BV} to L_D which, if indeed every L_D program is functional, produces equivalent programs. The proof that every L_D program is functional on the modified interpreter fills Chapters 4, 5, 6, and 7. Chapter 8 includes a judgment of how well the goal of maximal concurrency has been met.

Chapter 4

The Entry-Execution Model

This chapter introduces the entry-execution model of concurrent computation. The purpose of developing this model is to make the results of the thesis as widely applicable as possible. The major result is that a language L with Structure-as-Storage operations can be made a determinate language by modifying it according to a certain scheme. The entry-execution model of L focuses on just those aspects of L pertinent to this statement: whether it is determinate and whether it includes the Structure-as-Storage operations, in either a simple or a modified form. Details of L not germane to these issues are abstracted away.

Specifically, the results of this thesis can be applied to any language L and interpreter I on which L runs by the following procedure:

1. Modify I so that the outputs of Select operators are withheld in accordance with the Blocking Discipline. Restrict L to those programs in it which satisfy properties analogous to the Determinacy Condition and the Read-Only Condition. Call the modified interpreter I' and the restricted language L'. (This step has already been performed for the data-flow language L_{BS}.)

2. Construct an entry-execution model E of L' running on I'. The general form of such a model is defined below in Section 4.2; as an example, an algorithm for constructing a model of any data-flow language and interpreter is given in Section 4.3.

3. Check that E satisfies the constraints defining a Structure-as-Storage (S-S) model, given in Section 5.1. E is an S-S model iff L' contains operations having the same order-dependent behavior on I' as the structure operations in L_{BS}; this is proven in Section 5.3.

4. Check that E satisfies the Determinacy Axioms, given in Section 6.2. These are simple properties of the control portion of a program which are used to prove that the program is determinate. Most of them are used in existing proofs of determinacy for languages without structure operations, and so are well understood. One axiom asserts the key requirement of freedom from conflict between structure operations; E should satisfy this axiom if the modifications in Step 1 were made correctly. Then the principal theorem applies to E: An S-S model which satisfies the Determinacy Conditions is a determinate model (defined in Section 6.1).

5. Prove from the construction in step 2 that E is a determinate model only if L' running on I' is a functional language.

The final three steps are applied in Chapter 7 to the language L_{BS}.

The general form of an entry-execution model is given in Section 4.2 below; this is prefaced by a description of existing models of concurrent computation which shows their inappropriateness for the current research. Section 4.3 then provides an algorithm to construct an entry-execution model of any data-flow language and interpreter.

4.1 Historical Perspective

Several models of concurrent computation have been developed in the past ten years. Each of these different models was designed to aid in

the study of particular properties of parallel programs, usually
determinacy and equivalence. Certain details about a parallel program
have no bearing on its determinacy or equivalence, and so are treated
abstractly in the models. These models are very specific, however, about
those other details which have a strong impact on the issues of interest.
The basic elements common to all models for which determinacy is a meaning-
ful concept are described briefly below. Each of these elements is
characterized as to the degree of abstraction with which it is typically
treated, to show that the development of the entry-execution model has been
guided by the same principles as this previous work.

These models of concurrent computation are based on five concepts,
which are described below, both in general and by reference to the data-
flow model presented in Chapter 2.

1. A program contains (among other things) a set of <u>instructions</u>
    (actors). Each instruction specifies (among other things) an
    <u>operation</u>.

2. Computing by a program involves a set of <u>executions</u> (firings).
    An execution is the application of some instruction's operation
    to a set of input values to produce a set of output values. It is
    characteristic of concurrent computation that the relative order in
    which these executions occur is not totally fixed by the program.
    Instead, the program determines a set of possible relative orders
    or <u>computations</u> (firing sequences). A computation is a sequence of
    <u>events</u>, each of which is typically either the initiation or the
    termination of some execution; this allows modeling not only

different initiation orders, but also multiple executions in progress (initiated but not terminated) concurrently.

3. The instructions in a program are interconnected by a (local) memory structure. This consists of a set of memory elements (arcs). Each instruction is assigned some subset of these elements as its inputs and another subset as its outputs. Each time an instruction executes, input values are read from its input memory elements, and results are written into its output elements.

It is in this memory structure that the greatest diversity among models appears. Each element may store just one value, with either destructive (data flow) or non-destructive [24,28] readout. Alternatively, each element may be a first-in, first-out queue [2,23,34]. The interconnection of instructions and memory elements may be arbitrary or may be restricted, as in the case of data flow (in which each element is an input of at most one instruction and an output of at most one instruction).

4. Every program has a control portion. There is a set of states defined over the control portion, and a universal, non-deterministic state-transition rule. This rule defines a set of enabled events (initiations and terminations) for each possible state of the control. It also describes the new state resulting from each possible choice of which enabled event occurs next. The manner in which a state set and transition rule generates a set of possible computations is the same in all models, and is exemplified by data flow (Definition 2.1-5). The major difference among the models is in the representation of the state. The state may be embedded in

the memory structure, either in the amount of data in input queues
(computation graphs [23] or data flow), in auxiliary control infor-
mation stored in each element (program graphs [30]), in a combination
of these (graph programs [2]), or in the values of stored data
(computational schemata [28]). Alternatively, there may be a
separate control structure, consisting of a set of counters (flow
graph schemata [34] or parallel flowcharts [24]) or precedence graphs
[18,28], or the state set can be completely arbitrary (parallel
program schemata [24]).

5. There is a definition of a determinate program. The general notion
   may be stated as: Given a program and initial local memory content,
   every memory element has the same sequence of values written into it
   during all possible computations. Clearly, the exact definition
   depends on the particular memory structure unique to each model.

Out of the body of research employing these models have come several
general facts about determinacy in parallel programs. One of the most
significant of these is that determinacy is not affected by the particular
choice of operations performed by the instructions. The only requirement
is that all operations satisfy the following two properties:

   a. Determinism - The outputs of an execution of the operation depend
      only on the inputs to that execution.

   b. Finite delay - Once initiated, an execution of the operation must
      terminate within a finite time.

   It is significant that all of these models assume that any operation
used in a program satisfies these properties. As a consequence, most
models choose to abstract away the particulars of operations by defining

parallel schemata. A schema is a program with all instructions replaced
by operators. An operator differs from an instruction in that it has an
abstract operation symbol in place of a specific operation. That is,
where an instruction in a program has an operation like addition, the
corresponding operator in a schema might have the symbol 'f'. Study of
schemata has led to the discovery of sufficient conditions for their
determinacy (the Determinate Schema Axioms, presented in Section 6.2).
If a schema is determinate, then any program obtained by replacing each
abstract operation symbol with a specific operation is also determinate
(assuming that the specific operation is deterministic and has finite
delay).

This has been the main thrust of abstraction in the past: going from
concrete programs to schemata, which have concrete memory and control
structures but abstract operations. With one exception (noted shortly),
there has never been an attempt to instead abstract away both the memory
and control portions. It has apparently always been felt that it is more
challenging to verify the Determinate Schema Axioms for a particular
concrete memory/control structure than to verify determinism and finite
delay for a particular operation. Therefore, most of the previous models
were directed toward devising a general form for memory and control which
(1) is "practical", for programming and/or implementation, and (2) makes
it easy to identify the schemata in that form which satisfy the Determinate
Schema Axioms.

This thesis presents a different challenge in guaranteeing determin-
acy. It is assumed that programs will be written using any schema form
which has been (or will be) developed. This means that the problem of

identifying which programs' memory and control structures satisfy the Determinate Schema Axioms is not of interest. Rather, the concern here is for defining useful non-deterministic operations in such a way that it is still easy to identify determinate programs. This change in focus calls for a radically-different model of parallel programs, one with abstract memory and control, but concrete operations. What is unusual about this new entry-execution model is not the principle of abstraction, but the particular choice of aspects to be abstracted.

An increased emphasis on the definitions of operations is evident in the efforts of [3], [22], and [27] to specify the semantics of a schema language without using an interpreter. These researches defined an operator as a function from the vector of sequences of tokens appearing on its input arcs to a vector of sequences of tokens appearing on its output arcs (necessary to handle the gates, which do not always consume input tokens and produce output tokens). This new tool is not relevant to the problem under consideration here, however, because of the following two characteristics:

1. The concept was developed in an attempt to specify the (possibly-partial) function from program inputs to outputs realized by a program which is known to be functional and well-behaved.

2. It allowed defining operators for which the outputs of an execution depend on the sequence of past inputs to that operator (rather than on just the current inputs).

But the concern here is for deciding whether or not a program is function functional, given that it contains operators the outputs of which may depend on the sequence of past inputs to other operators as well.

Closest in spirit to the approach taken here is that of Greif in her thesis on the semantics of communicating parallel processes [19]; the similarities extend to the definition of a Structure-as-Storage model, and so a detailed comparison has been deferred to the end of Section 5.1. Her work was based on the "actor model", the only well-known effort to abstract away control and local-memory structures. Actors also, however, abstract away the concept of a program as a fixed set of instructions, which was undesirable for the present purposes.

4.2 Definition

An entry-execution model differs from a schema model in two major regards: (1) The abstract programs bear no resemblance to real programs. (2) Computations are sequences of events other than initiations and terminations of executions. These differences will be motivated here during the top-down definition of the general form of an entry-execution model.

The top level establishes the undefined concepts which will be needed:

Definition 4.2-1   An entry-execution model of a language is a five-tuple

$$(V, L, A, \text{In}, E)$$

where

        V is an atomic value domain

        L is a set of labels

        A is a domain of primitive actions

        In is a function assigning to each action in A an integer,

           its input arity

        E is a set of expansions, defined below

$\triangle$

The set V of values is arbitrary, but must be made explicit in order for the notion of determinacy to make sense. The model also retains the idea that a program contains instructions which have actions (e.g., Select, add, merge) associated with them. Each instruction in a program must be uniquely identified by a label from L. Each action associated with any instruction must be in the set A. Furthermore, an execution of instruction d must have a number of inputs equal to the input arity of the action associated with d. L is the only one of these entities which is abstract; the determinacy of a program does not depend on the particular labels on the instructions. It does however depend on the exact definition of at least some actions (the structure operations), as well as on their input arities. It is obvious how V, L, A, and In would be chosen in modeling a data-flow language.

4.2.1 The Abstract Programs

When the specific operations are abstracted away from a program, the result is a schema. When the specific memory and control structures are abstracted away from a program, the result is an expansion:

__Definition 4.2-2__ Given a model (V, L, A, In, E), each __expansion__ in E is an ordered pair (Int, J) where

> Int is an __interpretation__, an ordered triple $(\text{St}, I, \text{IE})$ in which
>> $\text{St} \subseteq L$,
>>
>> $I: \text{St} \to A$, and
>>
>> IE is a set of __executions__ (defined below),

and J is a set of <u>jobs</u> <u>for</u> <u>Int</u>, also defined below.

$\triangle$

An expansion retains none of the structure of a program that a schema does. This is the first distinctive feature of the entry-execution model. An expansion is as useful an abstraction as a schema is, however, as the following argues.

A parallel program P determines a set of jobs. "Job" here connotes the set of possible computations by P on a distinct program input. Thus each possible input to P gives rise (in principle) to a different job. In a schema model, each possible distinct input to P corresponds to a different equivalence class of initial states for the memory and control portion of P. Whenever desired, this compact initial-state representation can be expanded into a job, by generating all possible sequences of applications of the state-transition rule starting in one of those initial states.

In the entry-execution model, there is no concept of state, and hence no state-transition rule to apply. Jobs are still of interest, because ultimately determinacy is a property of jobs. But their derivation through specific state transitions is not of interest. Therefore, the details of memory and control are abstracted away from a program, leaving:

1. a set of instruction labels (St),

2. an association of actions with these labels (I),

3. a distinguished set of executions (IE) whose outputs will be used to model the program's inputs, and

4. the set of jobs resulting from expanding each equivalence class of initial states for the program (J).

This abstract program will be known here as an expansion. If an expansion is determinate, then any program, with any explicit memory and control structure, which yields that expansion is also determinate. This is the same spirit in which the determinacy of programs is implied by the determinacy of schemas.

The next definition clarifies the point that a job is not an arbitrary set of computations.

Definition 4.2-3 Given an interpretation Int, a job for Int is a set of computations for Int (defined later.)

△

Qualifying a computation as being "for Int", where Int = (St, I, IE), essentially just specifies that

1. all executions are of instructions having labels in St, and

2. each execution of an instruction labelled d has the proper number of inputs for the action I(d).

The conceptual significance of a job is that it represents all computations by a program on "the same input". Any precise character-ization of a job then necessarily makes reference to the set of input values of a program. In a standard schema model, these are just the contents of a designated subset of locations in the memory structure of an initial state. In the entry-execution model, however, there is no memory structure. Instead, these program inputs will be modeled as the outputs of certain executions: those in the designated set IE in the interpretation associated with the program. These executions will in

general be dummies, i.e., not "real" executions of instructions in the program. This artifice is best illustrated by example, as in the entry-execution model of a data-flow language (cf. Definition 4.3-1 below).

The determination of which sets of program input values constitute "the same input" to a program is highly language-dependent; consequently, including any further constraints here in the general definition of a job may render interesting languages incapable of being modeled. The argument for this claim is deferred until after the completion of the definition of a model.

4.2.2 The Computations

The second distinctive feature of the entry-execution model is its definition of a computation. The most important criterion in designing this is that there be a concise definition of determinacy for a set of computations. A crude expression of a suitable notion has already been given, in terms of the schema model of data flow, as the five Determinacy Assertions (Section 3.1.2). A principal source of clumsiness in those assertions was the frequent occurrence of the phrase "the jth firing of actor d"; consequently, the entry-execution model offers a more concise denotation:

<u>Definition 4.2-4</u> Given a model (V, L, A, In, E), an <u>execution</u> is an ordered pair consisting of a label $d \in L$ and a positive integer k, written

$$Ex(d,k)$$

$\triangle$

In an entry-execution model of a data-flow firing sequence, $Ex(d,k)$ denotes the kth firing of the actor labelled d.

The Determinacy Assertions also make reference to the value of a particular input to a particular firing, and to the direct transfer of that input from an output of some other firing. Appropriately, then, a computation in the entry-execution model is a sequence of entries, each representing the transfer of a single atomic value from an output of a source execution to an input of a destination execution:

Definition 4.2-5 Given a model (V, L, A, In, E), a source is an ordered pair consisting of an execution e and a positive integer i, written

$$\text{Src}(e,i)$$

A destination is an ordered pair consisting of an execution e and a positive integer j, written

$$\text{Dst}(e,j)$$

A transfer is an ordered pair (s,d), where s is a source and d is a destination.

An entry is an ordered pair consisting of a transfer and an atomic value from V. If f is an entry, then $T(f)$ denotes the transfer component of f and $V(f)$ denotes the value of f. Letting the transfer of f be

$$(\text{Src}(e_1,i), \text{Dst}(e_2,j))$$

f is an output entry of execution e_1, and is the jth input entry of execution e_2. The target execution of f is e_2.

$\triangle$

The appearance of an entry with transfer

$$(\text{Src}(\text{Ex}(d_1,k_1),i), \text{Dst}(\text{Ex}(d_2,k_2),j))$$

and value v in a computation modelling a firing sequence Ω means that:

The value of the number-j input to the k_2^{th} firing of actor d_2 in Ω was v, and was produced as the number-i output of the k_1^{th} firing of d_1.

The set of entries constituting a computation must satisfy certain obvious constraints in order to be a reasonable model:

Definition 4.2-6 Given a model (V, L, A, In, E) and an interpretation Int = (St, I,IE) in some expansion in E, a computation for Int is a (possibly infinite) sequence of entries

$$\omega = f_1, f_2, \ldots$$

satisfying the following:

1. Let $e = Ex(d,k)$ be any execution of which there is either an input entry or an output entry in ω. Then $d \in ST$, and there are at most In($I(d)$) input entries to e in ω. If ω contains exactly In($I(d)$) input entries of e, then e is initiated in ω (with respect to Int), and the last such input entry in ω is the initiating entry of e.

2. The destinations of the transfers of the entries in ω are all distinct (i.e., for each j, an execution has at most one number-j input entry in ω).

3. For any source s, denote by OE_ω(s) the set of entries in ω whose transfers have source s. Then all entries in OE_ω(s) have the same value. This common value is the value of source s (in ω).

$$\triangle$$

Of the five Determinacy Assertions, only the third and fifth concern pointers or structure operations; thus the first, second, and fourth together define determinacy of a program having no structure operators. The following statement, which is as strong as those three assertions, illustrates the conciseness of expression possible in the entry-execution model:

> Given an expansion (Int,J), for any $J \in J$, every halted computation
>
> in J contains exactly the same set of entries.

This is the definition of a determinate expansion in the absence of

structure operators (the complete definition may be found in Section 6.1).

It is claimed, without proof, that any data-flow program P whose expansion

is determinate must satisfy the five Determinacy Assertions; it is proven

(in Chapter 7) that P is at least functional. The ease of defining

determinacy illustrates the benefits of choosing entries as the events in

computations, which choice was the second major departure from the schema-

model norm.

It has been claimed that an exact description of the set of compu-

tations by a single program on a single input (i.e., of a job) is highly

language-dependent. This is easily seen by comparing the appropriate

descriptions for data-flow languages with and without structure operators.

In an entry-execution model of any data-flow language (as constructed in

Section 4.3), the value of the token on each program input arc is repre-

sented as the value of a distinctive, fixed entry; call the entry repre-

senting the number-i program input the "number-i program input entry".

In the model of a data-flow language without structure operators (such

as L_B), a job is easily characterized: Two program inputs are equal iff

they are identical; hence, for any i, the number-i program input entries

in all computations in a job must have the same value. In the model of a

language with structure operators, however, this constraint applies only

to those program input entries whose values are not pointers. The values

of pointer-valued program input entries are arbitrary, as shown next.

Letting p be any pointer which is on a program input arc in some initial state, for any other pointer q there is an equal initial state in which that arc has q on it. Those two initial states represent the same program input. The set of all computations by that program on that input is a job J. Therefore, there will be in J some computations in which the corresponding program input entry has value p, and others in which that entry has value q. Since this statement is true for any pointers p and q, every pointer appears as the value of that program input entry in some computation in J. Thus it is seen that any attempt to develop a non-trivial characterization of "the same input" which is valid in the models of all interesting languages is ill-advised.

This completes the presentation of the general form of an entry-execution model of a programming language. This model was motivated by the desire to make the results of the thesis applicable to as wide a range of languages as possible. To this end, the memory and control portions are abstracted away from a program, to focus on the definitions of the operations. The resulting abstract program looks radically different from a schema, in ways which have been pointed out.

The merits of this model can be judged only on the basis of (1) how easily results are stated in its terms, and (2) how easily they are then applied to different languages. Evidence on the first of these issues may be found in Chapters 5 and 6, and on the second in Chapter 7. The remaining section of this chapter constructs a model of data-flow languages. The current section now concludes with some useful properties of entry-execution models and an algorithm for producing pictorial representations of computations.

4.2.3 Properties

The following defines two properties which will be assumed to hold for all models of interest.

Definition 4.2-7 A computation ω is <u>causal</u> (<u>with respect to interpretation Int</u>) iff the following is true for any execution e: For any prefix αf of ω in which f is an output entry of e, e is initiated in α (wrt Int).

A computation ω in a job J is <u>halted in J</u> iff it is a proper prefix of no other computation in J. A job J has the <u>Prefix Property</u> iff for every ω in J, every prefix of ω is in J.

$\triangle$

Causality, while not strictly essential, greatly simplifies the proofs developed later; it is proven shortly that all computations in a model of data flow are causal. Similarly, a job could include only halted computations. But the Prefix Property allows writing, for example, "if ω is in J, then so is ωf," instead of "if ω is a prefix of some computation in J, then so is ωf." For convenience, then, both causality and the Prefix Property are assumed in the general proof of determinacy in Chapter 6.

Finally, the following notational conventions will be observed:

1. Roman letters (f, g, h, k) will be used to denote single entries, while Greek letters (ω, α, β, ...) will be used to denote sequences of zero or more entries.

2. Given a computation ω, $\text{Ent}_{\omega}(e,j)$ denotes that unique entry in ω whose transfer has destination $\text{Dst}(e,j)$. When ω is understood, the subscript may be omitted.

3. Given an interpretation $(\text{St}, I, \text{IE})$, execution $\text{Ex}(d,k)$ for any $d \in \text{St}$ and any k is <u>an execution of</u> the action $I(d)$.

4.2.4 Pictorial Representation

The relationships among the entries and executions of a computation can be depicted as a directed graph. The nodes of the graph represent executions and the branches represent entries. The branches terminating on a node n represent the input entries of the execution e represented by n, and the branches leaving n represent e's output entries.

__Algorithm 4.2-1__ To construct an __entry-execution graph__ for computation ω:

1. Initialize an entry counter EC to 1.

2. For each entry f in ω in order: Let $T(f) = (Src(e_1,i), Dst(e_2,j))$ and let $V(f) = v$.

   a) If there is no node labelled with e_1 (or e_2) in the graph yet, add an open figure (e.g. a circle) with e_1 (or e_2) written inside it.

   b) Draw a directed branch from the node labelled with e_1 to the node labelled with e_2. Write i beside the tail of this branch and j beside its head.

   c) Label the branch (distinctively) with atomic value v and with EC. Increment EC by 1.

$\triangle$

4.3 An Entry-Execution Model of Data-Flow Languages

This section first presents an algorithm for deriving the entity EE(L,I) from any data-flow language L and interpreter I. It then proves some important properties of EE(L,I), including that it is indeed an entry-execution model. This serves two purposes: (1) it is a specific example of the construction of a model, and (2) it is the first step in applying the results of the thesis to the data-flow language L_D on the modified interpreter M. The latter process was outlined at the start of this chapter. The algorithm presented here figures prominently in several steps of the proof: It is used in Chapter 7 to prove that $EE(L_D,M)$ is a Structure-as-Storage model satisfying the Determinacy Axioms. The result of Chapter 6 then applies, saying that every expansion in $EE(L_D,M)$ is determinate. Finally, the algorithm is used to prove, from this result, that every program in L_D is functional when run on M.

4.3.1 The Construction of EE(L,I)

The steps in constructing EE(L,I) are first presented informally:

1. For each initial state S of a program P, and firing sequence Ω starting in S, construct the canonical computation $\eta(S,\Omega)$, using Algorithm 4.3-1 below. In $\eta(S,\Omega)$, there is an entry for each token appearing on an arc in P in the course of Ω, and the entries are arranged in the order of the removal of their corresponding tokens.

2. Construct $J_{S,\Omega}$ as a constrained set of permutations of $\eta(S,\Omega)$. $J_{S,\Omega}$ contains all halted computations which model, in a sense described later, the firing sequence Ω.

3. Construct the set consisting of all prefixes of all computations in $J_{S,\Omega}$. Every computation in this set models Ω, and the set satisfies

the Prefix Property.

4. Repeat steps 1, 2, and 3 for all firing sequences Ω starting in all
   initial states in the equivalence class E containing S. The job
   J_E is the union of all the sets of computations produced in this
   manner.

5. Repeat steps 1 through 4 for all equivalence classes of initial
   states of program P. The set J of jobs produced, together with an
   interpretation for P, is the expansion corresponding to P.

6. Repeat step 1 through 5 for all programs in the language L. This
   generates the set of expansions E, which, together with appropriate
   domains of values and actions, constitutes EE(L,I).

The formal definition of EE(L,I) is presented next, in a top-down
fashion paralleling the general description of an entry-execution model.
Each definition given below is followed by an explanation.

<u>Definition 4.3-1</u> Given a data-flow language L and interpreter I, EE(L,I)
is the five-tuple

$$(V, L, A, \text{In}, E)$$

where

   V is the atomic value domain of L

   L = W$\cup$DL, where W is the universe of labels in L, and

       DL = {"ID", "IT", "IF"}$\cup$(W$\cup${"OD"})$\times$N

           where N is the set of natural numbers, and none of

               "ID", "IT", "IF", and "OD" is in W

   A is the set of actor types in L, plus the distinctive IG and OA

       actions (described below).

In assigns zero to IG, one to OA, and to every other actor type in

A assigns the number of input tokens which that type removes

at each firing

E is the set containing, for each program P in L, the expansion of

P, defined below.

$$\triangle$$

IG is the initial-value generating action, and OA is the output-accepting

action. These are distinctive in that they are not associated with any

actors in any program in L. IG is needed because, as noted earlier,

initial values in the memory structure of a program must be modeled as

the outputs of executions. Since in data flow, these values are not

outputs of real executions, dummy executions must be created. The dummy

executions Ex(IT,0), Ex(IF,0), and Ex(ID,0) will act as sources of initial

true, false, and program input tokens respectively; consequently, these

three executions constitute the set IE in all interpretations. Each of

these is an execution of the distinctive action IG. Since In(IG) = 0,

none of these three dummy executions will have any input entries.

OA is the action of another set of dummy executions. These will be

used to model the program output tokens. Each execution of the OA action

will be Ex((c,j),0), where c is either the label of an actor or the

distinctive label "OD", and j>0. These composite labels (c,j) allow

associating a unique such execution with every arc b in a program, by

the following correspondence:

If b is the number-j input arc of the actor labelled d, then the

associated dummy execution is Ex((d,j),0).

Otherwise, b is the number-j program output arc, for some j, in which

case the associated dummy execution is Ex((OD,j),0).

Since In(OA) = 1, each such execution e will have exactly one input entry

in all computations, with destination Dst(e,1).

As will be seen, every entry in a computation whose target is not

one of these dummy executions models the removal of some unique token by

a firing of a real actor. Without these dummy executions, therefore,

there would be no entries modeling the tokens left in the final state

after a halted firing sequence. But these are just the tokens which

matter in determining if two such final states are equal, i.e., if the

program is functional. Having these added entries makes it much easier

to prove that only a functional program can give rise to a determinate

expansion.

The destinations in the transfers of the entries modeling tokens

left in a final state are all distinct, as required. This is done by

making each such destination be Dst(e,1) where e is a unique execution

of the OA action. It may seem that a neater choice would be to have each

destination be Dst(e,j) where e is a common execution of OA but the

integers j are distinct. This is not possible for two reasons:

1. In order to associate each distinct destination with an arc, a

   program would have to include a numbering of all the arcs, which

   it does not.

2. There would be an indefinite number of input entries to the common

   execution e, violating the requirement that there is a maximum

   number In(OA) of such entries in all computations.

In general, an action has an input arity equal to the number of input arcs of any actor with which it is associated. The only exception is the merge gate, which always removes two tokens from its three input arcs; its input arity is therefore two.

<u>Definition 4.3-2</u> (Expansion of P) Given a data-flow language L, let P be any program in L. Then the <u>interpretation</u> <u>of</u> <u>P</u>, Int(P), is (St,I,IE),

where   St is the set of labels of the actors in P, plus the label set DL

I: St $\rightarrow A$ assigns to each label of an actor in P the type of that actor, assigns the action IG to each of the labels "ID", "IT", and "IF", and to every other label in DL assigns the action OA

IE $=$ {Ex(ID,0), Ex(IT,0), Ex(IF,0)}

The expansion of P is the ordered pair (Int(P),J) in which J is the set of jobs

$$J = \{J_E \mid \text{E is an equivalence class of initial states for P}\}$$

where J_E is the <u>job</u> <u>for</u> <u>E</u>, defined below.

$\triangle$

J is the set of jobs resulting from expanding all initial states of some program. Thus the ordered pair (Int(P),J) is the type of abstract program being called an expansion.

<u>Definition 4.3-3</u> Given an equivalence class E of initial states for a data-flow program P, the <u>job</u> <u>for</u> <u>E</u>, J_E, is given by

$$J_E = \bigcup_{S \in E} \bigcup_{\Omega \in FS(S)} \pi(J_{S,\Omega})$$

where FS(S) is the set of all halted firing sequences starting in S

π takes a set of computations into the set of all their prefixes

$J_{S,\Omega}$ is the <u>set</u> <u>of</u> <u>computations</u> <u>for</u> S <u>and</u> Ω, defined below.

$\triangle$

$J_{S,\Omega}$ is a constrained set of permutations of the <u>canonical</u> <u>computation</u> $\eta(S,\Omega)$. The algorithm for constructing $\eta(S,\Omega)$ is given next, followed by the definition of $J_{S,\Omega}$.

<u>Algorithm 4.3-1</u> Given an initial state S of a data-flow program P and a firing sequence Ω starting in S, this algorithm constructs the <u>canonical</u> <u>computation</u> $\eta(S,\Omega)$ in two steps. The first step is to recursively construct the computation $\omega(S,\Omega)$ as follows:

Basis: $\omega(S,\lambda) = \lambda$.

Induction step: For firing sequence $\Omega\varphi$, in which the last firing φ is of the actor labelled d in P, $\omega(S,\Omega\varphi)$ is derived from $\omega(S,\Omega)$ as follows:

Let $e = Ex(d,n)$, where $\Omega\varphi$ has exactly n firings of d. Let a_1, a_2, ..., a_m be the input arcs of d from which tokens are removed in going from state $S\cdot\Omega$ to $S\cdot\Omega\varphi$, arranged in the order imposed on them by P. Then $\omega(S,\Omega\varphi)$ is the concatenation

$$\omega(S,\Omega\varphi) = \omega(S,\Omega),f_1,f_2,\ldots,f_m$$

where each entry f_k, $k = 1,\ldots,m$, is specified by:

$V(f_k)$ is the value of the token removed from arc a_k (except that if that value is tagged pointer (p,R) or (p,W), $V(f_k)$ is just p).

The destination of the transfer $T(f_k)$ is Dst(e,j), where a_k is the

number-j input arc of d.

The source of $T(f_k)$ is given as the value of the function

Source(a_k, S, Ω), which is defined next.

The value of Source(a, S, Ω) for any arc a and firing sequence Ω starting in state S depends on whether or not the token on a in $S \cdot \Omega$ was on that arc in S:

1. If it was, then a is either a program input arc or a control arc.

    a. If a is the number-i program input arc of P, then

    Source(a, S, Ω) = Src(Ex(ID,0),i).

    b. If a is a control arc, then Source(a, S, Ω) = Src(Ex(IT,0),1)

    or Source(a, S, Ω) = Src(Ex(IF,0),1), according to whether the

    token is a _true_ or a _false_ token.

2. Otherwise, let i be such that a is in the number-i group of output

    arcs of actor d' in P. Then Source(a, S, Ω) = Src(Ex(d',n'),i),

    where there are exactly n' firings of d' in Ω.

Now $\eta(S, \Omega)$ is defined as:

If Ω is not halted, then $\eta(S, \Omega)$ = $\omega(S, \Omega)$.

If Ω is halted, let b_1, b_2, ..., b_r be the arcs of P which hold

tokens in the final state $S \cdot \Omega$. Then $\eta(S, \Omega)$ is the concatentation

$$\eta(S, \Omega) = \omega(S, \Omega), g_1, g_2, \ldots, g_r$$

where each entry g_h, h = 1,...,r, is specified by:

$V(g_h)$ is the value of the token on arc b_h in the final state

(except that if that value is (p,R) or (p,W), $V(g_h)$ is just p).

The destination of the transfer $T(g_h)$ is Dst(e,1), where

execution e depends on whether b_h is a program output arc of P:

1. If b_h is the number-j output arc, for some j, then
   $e = Ex((OD,j),0)$.

2. Otherwise, b_h is the number-j input arc of an actor
   labelled d in P, for some j, and $e = Ex((d,j),0)$.

The source of $T(g_h)$ is $Source(b_h,S,\Omega)$.

$\triangle$

The computation $\omega(S,\Omega)$ is simply Ω with each firing φ replaced by a
set of entries describing the tokens removed by φ. If φ is the nth firing
of the actor labelled d, these entries all have as a common target the
execution $Ex(d,n)$ referring to φ. Let a_k be an arc from which a token was
removed by φ, and let d' be the actor of which a_k is an output arc. Then
the token removed from a_k by φ was placed there by the firing of d' in Ω
which most recently preceded φ. The execution referring to that firing of
d' is $Ex(d',n')$, where exactly n' firings of d' precede φ in Ω. That
execution is the source of the entry describing the token removed from a_k
by φ.

The canonical computation $\eta(S,\Omega)$, for a halted firing sequence Ω,
supplements $\omega(S,\Omega)$ with a set of entries describing the tokens left in $S \cdot \Omega$.
For each arc b holding such a token, there is an entry whose source is the
execution referring to the firing which placed that token on b. The target
of that entry is the unique dummy output execution associated with b.
Therefore, for each token which appears in the course of Ω, there is a
uniquely-identifiable entry f in $\eta(S,\Omega)$, and V(f) equals the value of that
token; this is true even if that token is not removed by any firing in Ω.

The canonical computation $\eta(S,\Omega)$ retains almost all of the information
contained in the original firing sequence Ω. What cannot be conveyed by a

satisfies all of the following (given Int(P)):

1. $\Phi(\beta)$ is the reduction of Ω.

2. β is causal.

3. Let αf be any prefix of β, let θ be the prefix of Ω whose reduction is $\Phi(\alpha)$, and let the destination in T(f) be Dst(Ex(d,k),j).

    a. If $d \notin DL$, then let b be the number-j input arc of the actor labelled d. That actor is enabled in $S \cdot \theta$, and if it is a merge gate and b is its T (F) input arc, then its control input arc holds a <u>true</u> (<u>false</u>) token in $S \cdot \theta$.

    b. If $d \in DL$, then d = (c,n). Let b be the number-n program output arc of P, if c = "OD", or else the number-n input arc of the actor labelled c. Then there is a token on b in $S \cdot \theta$, and if c labels an actor, there is no firing sequence starting in $S \cdot \theta$ which contains a firing of c.

$\triangle$

The last constraint in this definition is necessary to make tractable the proof of the key property of persistence. A full discussion of its significance is provided in conjunction with that proof in Chapter 7; suggestions for more meaningful alternative specifications are given in Chapter 8.

This completes the definition of EE(L,I); a proof that EE(L,I) is an entry-execution model will be given shortly. First, this example of a specific model can be used to gain an appreciation for the choice of entries as the events in computations.

One benefit of using entries has already been illustrated by the particularly compact definition of a determinate expansion. Recalling the description in Section 4.1 of earlier models, determinacy usually was

computation are the essentially-arbitrary pointer-node pairs in the Copy

firings in Ω. The consequences of this loss of information are explored

at length in Section 5.2.1. Here it will just be shown that from $\eta(S,\Omega)$

can be reconstructed the reduction of Ω (recall from Definition 2.4-5 that

the reduction of Ω is Ω without the pointer-node pairs).

Definition 4.3-4 Given a computation α and an interpretation Int, the
firing sequence reconstructed from α with respect to Int, $\Phi(\alpha)$, is defined
recursively as follows:

Basis: $\Phi(\lambda) = \lambda$.

Induction step: $\Phi(\alpha f)$ depends on whether or not entry f is the initiating

entry in αf (wrt Int) of an execution $e = Ex(d,k)$ for any $d \notin DL$ and any k.

If not, then $\Phi(\alpha f) = \Phi(\alpha)$. If so, then $\Phi(\alpha f) = \Phi(\alpha)\varphi$, where φ is the

firing which is just the label d.

$\triangle$

In $\eta(S,\Omega)$, the input entries of an execution are all grouped together.

If the nth firing in Ω is the kth firing of actor d, then the nth such

group in $\eta(S,\Omega)$ are input entries to $Ex(d,k)$. This execution is then the

n^{th} initiated in $\eta(S,\Omega)$, and so the nth firing in $\Phi(\eta(S,\Omega))$ is the kth

firing of d. Hence, $\Phi(\eta(S,\Omega))$ is the reduction of Ω.

For any permutation β of $\eta(S,\Omega)$ which preserves initiation order of

executions, $\Phi(\beta)$ will also be the reduction of Ω. The set of all such

permutations which are causal forms the basis of $J_{S,\Omega}$:

Definition 4.3-5 Let S be any initial state for a data-flow program P, and

let Ω be any halted firing sequence starting in S. Then the set $J_{S,\Omega}$ of

computations for S and Ω consists of each permutation β of $\eta(S,\Omega)$ which

defined as the equivalence of all the event sequences in a job, where each

event typically was either the initiation or the termination of an execu-

tion. Different models had different notions of equivalence, but there

was a very common method of proving determinacy. This technique, which

has been adapted for use here in Chapter 6, involves a great deal of

manipulation of event sequences:

The Determinacy Proof Technique - Prove that any pair of event sequences

in a job is equivalent by transforming one into the other through

a series of transpositions of adjacent events. Prove that each of

the transpositions takes a sequence ω_1 in the job into another

sequence ω_2 in the job which is equivalent to ω_1.

Event sequences are general enough that they could have been selected as

the representation of computations in the entry-execution model. The cost

of such a choice would have been that of having to manipulate an auxiliary

tabulation of the input and output values of each execution (in order to

define equivalence in the absence of explicit memory structure).

EE(L,I) demonstrates the definition of a canonical entry sequence

corresponding uniquely to a (reduced) event sequence. It will always be

possible to construct such a canonical entry sequence. There will always

be a method analogous to Definition 4.3-4 to reconstruct the unique event

sequence from each canonical computation. The cost of using entry

sequences as computations is that of having to reconstruct reduced event

sequences at each transposition in the determinacy proof. This was judged

to be the more efficient alternative.

This argues for defining a job J_E in EE(L,I) as the set of canonical

computations derived from all possible firing sequences starting in all

initial states in E. But J_E actually contains many non-canonical permutations of these computations. This is because transforming one canonical computation into a different one requires a complex permutation, dependent on the exact form of canonical computation peculiar to a model. To base a determinacy proof on such a complex permutation would defeat the goal of applicability to a wide range of models. The only elementary permutation which can be used repetitively to take any entry sequence into any other is a transposition of two adjacent entries. This imposes the requirement on a model that any computation in a job can be transformed into any other by a series of transpositions such that all intermediate computations are also in that job. Accordingly, jobs in EE(L,I) are augmented as in Definition 4.3-5.

4.3.2 Properties of Models of Specific Data-Flow Interpreters

The foregoing has defined the entity EE(L,I) for any data-flow language L run on any interpreter I. The rest of the thesis is concerned with the models of various data-flow languages run on two specific interpreters; the following notation will be used to differentiate between these interpreters:

For any data-flow language L:

EE(L,S) denotes the model of L run on the standard interpreter, and

EE(L,M) denotes the model of L run on the modified interpreter.

This concluding sub-section develops important general properties common models of languages run on either of these interpreters.

The first property is that, for any language L, EE(L,S) and EE(L,M)

are indeed entry-execution models. The only non-trivial proof

required is that, for every program P, every computation in every job from the expansion of P is a computation for Int(P). Since every such computation is a prefix of a permutation of one of a certain set of canonical computations, it is shown first that all of those canonical computations are computations for Int(P). This proof requires two preliminary results, of wide applicability, which are separated out as Lemma 4.3-1 next.

Given the canonical computation $\eta(S,\Omega)$ for any initial state S and firing sequence Ω starting in S, the following is apparent from Algorithm 4.3-1: the appearance in $\eta(S,\Omega)$ of an entry with value v whose transfer has destination Dst(Ex(d,k),j) means that the kth firing of d removed a token of value v from d's number-j input arc. Lemma 4.3-1 proves a symmetric statement about the significance of an entry with value v whose transfer has source Src(Ex(d,k),j). Usually, this means that tokens of value v were placed on the number-j group of output arcs of d at d's k^{th} firing. However, this may not be strictly true if d is a Select and those outputs were withheld on the modified interpreter; hence the weaker assertion of the Lemma below. Also shown is how the number of input entries to Ex(d,k) in $\eta(S,\Omega)$ is related to the number of firings of d in Ω.

Lemma 4.3-1 Let S be any initial standard or modified state for a program P, let Ω be any firing sequence starting in S, and let Int(P) be (St,I,IE). Then:

A: For any entry f in $\eta(S,\Omega)$, let the source in T(f) be Src(Ex(d,k),i), and let V(f) be v. If d is in St-DL, then there is a prefix $\Delta\varphi$ of Ω containing exactly k firings of d such that tokens of value v, (v,R), or (v,W) appear on the number-i group of output arcs of the

actor labelled d in P at the transition from $S \cdot \Delta$ to $S \cdot \Delta\varphi$.

B: For any execution $d = Ex(d,k)$, if d is in St-DL, then the number of input entries to e in $\eta(S,\Omega)$ or in $\omega(S,\Omega)$ is given by

$$\begin{cases} 0 & \text{if there are fewer than k firings of d in } \Omega \\ In(I(d)) & \text{otherwise} \end{cases}$$

Proof:

(1) Let b be any arc in P, and let θ be any prefix of Ω such that there is a token of value X on b in $S \cdot \theta$. Assume b is in the number-i group of output arcs of actor d and there are exactly k>0 firings of d in θ. Let $\Xi\varphi_d$ be the prefix of Ω in which φ_d is the kth firing of d. Then $\Xi\varphi_d$ is a firing sequence starting in S, and so d is enabled in $S \cdot \Xi$ Def. 2.3-1

(2) There is no token on any output arc of d in $S \cdot \Xi$ (1)+Defs. 3.3-6+2.1-4

(3) There is a prefix $\Delta\varphi$ of θ longer than Ξ - i.e., containing exactly k firings of d - such that tokens of value X appear on the number-i group of output arcs of d at the transition from $S \cdot \Delta$ to $S \cdot \Delta\varphi$ (1)+(2)

Now prove that A and B are true with $\omega(S,\Omega)$ substituted for $\eta(S,\Omega)$, by induction on the length of Ω.

Basis: $|\Omega| = 0$.

(4) $\omega(S,\Omega) = \lambda$, which has zero entries   Alg. 4.3-1

(5) A is vacuously true   (4)

(6) For any $d \in$ St-DL and k>0, there are fewer than k firings of d in Ω, and there are zero input entries to Ex(d,k) in $\omega(S,\Omega)$, hence B   (4)

Induction step: Assume A and B are true for $\omega(S,\Omega)$ if $|\Omega| = n$ and consider $\Omega = \theta\varphi$, of length n+1, in which the last firing φ is of actor c.

Let $\alpha = \omega(S,\theta)$ and $\beta = \omega(S,\theta\varphi)$.

(7) Let f be any entry which is in β but not in α, let the source in
   T(f) be $Src(Ex(d,k),i)$, and let V(f) be v. If $d \in St-DL$, there is
   a token of value v, (v,R), or (v,W) on an arc in the number-i
   group of output arcs of d in $S \cdot \theta$, and there are exactly $k > 0$
   firings of d in θ Alg. 4.3-1

(8) A is true for f   (7)+(1)+(3)

(9) Since A is true for all f in α, A is true for all f in β

                                                    (8)+ind. hyp. A

(10) Let j be such that there are exactly j firings of c in $\theta\varphi$. Then
   β is α followed by m input entries to $e = Ex(c,j)$, where m is the
   number of c's input arcs from which tokens are removed in the
   transition from $S \cdot \theta$ to $S \cdot \theta\varphi$ Alg. 4.3-1

(11) For any $d \in St-DL$ and $k > 0$, $(d \neq c \lor k \neq j) \Rightarrow$ there are fewer than k
   firings of d in θ iff there are fewer than k firings of d in $\theta\varphi$ $\land$
   there are the same number of input entries to $Ex(d,k)$ in β as
   in α (10)

(12) $\Rightarrow$ B for d and k   ind. hyp. B

(13) There are fewer than j firings of c in θ but not in $\theta\varphi$ (10)

(14) There are 0 input entries to e in α (13)+ind. hyp. B

(15) There are m input entries to e in β (10)+(14)

(16) $m = In(l(d))$ (10)+Defs. 4.3-1+4.3-2

(17) B for $\omega(S,\theta\varphi)$ (12)+(13)+(15)+(16)

Thus it is proven that A and B are true for $\omega(S,\Omega)$ for any Ω. Now prove

that A and B are true for $\eta(S,\Omega)$. If Ω is not halted, then

$\eta(S,\Omega) = \omega(S,\Omega)$. Assume therefore that Ω is halted.

(18) Let $\alpha = \omega(S,\Omega)$ and $\beta = \eta(S,\Omega)$. Let f be any entry in β, let

Src(Ex(d,k),i) be the source in T(f), and let V(f) = v. If f is

in α, then A is true by the above. Assume therefore that f is not

in α. If dϵSt-DL, then there is a token of value v, (v,R), or

(v,W) on an arc in the number-i group of output arcs of the actor

labelled d in $S \cdot \Omega$, and there are exactly k firings of d in Ω

Alg. 4.3-1

(19) A for f, hence for all entries in $\eta(S,\Omega)$ (18)+(1)+(3)

(20) For any execution Ex(d,k), if dϵSt-DL, then the number of input

entries to the execution in β is the same as in α. By the above,

that number is 0 if there are fewer than k firings of d in Ω, or

In(I(d)) otherwise   Alg. 4.3-1

$\triangle$

Now it can be shown that the canonical computations used to generate

the computations in the expansion of P are all computations for Int(P).

Also important to note is that any canonical computation is causal. The

proof of these properties is not very enlightening, and so has been

relegated to Appendix C.

Lemma 4.3-2 Let S be any initial standard or modified state of any

program P, and let Ω be any firing sequence starting in S. Then $\eta(S,\Omega)$

is a causal computation for Int(P).

$\triangle$.

For every computation α in a job from the expansion of program P,

there is an initial state S for P and a firing sequence Ω starting in S

such that the set of entries in α is a subset of those in $\eta(S,\Omega)$. Thus

it is easy to show from the above that α is a computation for Int(P), and that therefore:

Theorem 4.3-1 Given any data-flow language L, both EE(L,S) and EE(L,M) are entry-execution models.

Proof:

(1) The entities V, L, A, and In in both EE(L,S) and EE(L,M) satisfy the corresponding specifications for a model   Defs. 4.3-1+4.2-1

(2) Let E be the set of expansions from either EE(L,S) or EE(L,M), and let (Int,J) be any ordered pair in E. Then (Int,J) corresponds to a program P in L   Def. 4.3-1

(3) Int = Int(P), which is an interpretation   Defs. 4.3-2+4.2-2+4.3-1

(4) Let J be any job in J. Then there is an equivalence class E of initial (standard or modified) states for P such that $J = J_E$

Def. 4.3-2

(5) Let α be any computation in J. Then there is an initial (standard or modified) state $S \in E$ and a halted firing sequence Ω starting in S such that α is a prefix of some β in $J_{S,\Omega}$ Def. 4.3-3

(6) β is a permutation of $\eta(S,\Omega)$, so the set of entries in α is a subset of the set of entries in $\eta(S,\Omega)$ (5)+Def. 4.3-5

(7) $\eta(S,\Omega)$ is a computation for Int(P) = Int. Let Int = (St, I,IE)

(5)+Lemma 4.3-2

(8) Let $e = Ex(d,k)$ be any execution of which there is either an input or an output entry in α. Then there is an input or output entry of e in $\eta(S,\Omega)$ (6)

(9) $d \in St$ and there are at most $In(/(d))$ input entries to e in $\eta(S,\Omega)$,

    hence in α (8)+(7)+(6)+Def. 4.2-6

(10) The destinations of the transfers of the entries in α are all

     distinct, and all entries whose transfers have the same source

     have the same value   (7)+(6)+Def. 4.2-6

(11) α is a computation for Int   (8)+(9)+(10)+Def. 4.2-6

(12) J is a job for Int   (5)+(11)+Def. 4.2-3

(13) (Int,J) is an expansion   (4)+Def. 4.2-2

(14) E is a set of expansions, so EE(L,S) and EE(L,M) are entry-execution

     models   (2)+(13)+(1)+Def. 4.2-1

$$\triangle$$

    It was argued briefly earlier that the firing sequence reconstructed from any canonical computation $\eta(S,\Omega)$ is the reduction of Ω. The following Lemma provides a rigorous demonstration of this. It then uses that result to make explicit the tacit assumption that a job contains all of the canonical computations used to generate it.

Lemma 4.3-3 For any data-flow program P, let S be any initial standard or modified state for P, let Ω be any halted firing sequence starting in S, and let Int(P) be (St,$/$,IE). Then the firing sequence reconstructed from $\eta(S,\Omega)$ wrt Int(P), $\Phi(\eta(S,\Omega))$, is the reduction of Ω and $\eta(S,\Omega)$ is in $J_{S,\Omega}$.

Proof: All initiations and reconstructions are with respect to Int(P).

 (1) Let $\beta = \eta(S,\Omega)$ and let $\alpha = \eta(S,\Omega)$. Then β is a permutation of

    $\eta(S,\Omega)$ and β is α followed by input entries to executions in the

    set $\{Ex(d,k) \mid d \in DL\}$ Alg. 4.3-1+Def. 4.3-1

(2) $\Phi(\beta) = \Phi(\alpha)$ (1)+Def. 4.3-4

Prove by induction on the length of the prefixes θ of Ω that $\Phi(\alpha)$ is the reduction of Ω.

Basis: $|\theta| = 0$.

(3) $\omega(S,\theta) = \lambda$, the empty computation   Alg. 4.3-1

(4) $\Phi(\omega(S,\theta)) = \lambda$, which is the reduction of θ (3)+Defs. 4.3-4+2.4-5

Induction step: Assume that for the length-n prefix θ of Ω, $n \geq 0$, $\Phi(\omega(S,\theta))$ is the reduction of θ, and consider prefix $\theta\varphi$ of length n+1, in which the last firing φ is of actor d.

(5) Let $\delta = \omega(S,\theta)$ and $\gamma = \omega(S,\theta\varphi)$. Then γ is δ followed by m input
      entries to Ex(d,k), where φ removes m tokens   Alg. 4.3-1

(6) $m = \text{In}(I(d))$ (5)+Defs. 4.3-2+4.3-1

(7) Exactly one entry which is in γ but not in δ is the initiating entry
      of an execution, and that execution is Ex(d,k)   (5)+(6)+Def. 4.2-6

(8) $\Phi(\gamma)$ is $\Phi(\delta)\varphi'$, where φ' is the label d   (7)+Def. 4.3-4

(9) $\Phi(\delta)$ is the reduction of θ

(10) The reduction of $\theta\varphi$ is the reduction of θ followed by a firing
      which is the label d   Def. 2.4-5

(11) $\Phi(\gamma)$ is the reduction of $\theta\varphi$ (8)+(9)+(10)

Thus it is proven inductively that

(12) $\Phi(\eta(S,\Omega)) = \Phi(\beta) = \Phi(\alpha)$ is the reduction of Ω (1)+(2)

(13) β is causal wrt, and is a computation for, Int(P)   (1)+Lemma 4.3-2

(14) Let γf be any prefix of β in which $f = \text{Ent}(e,j)$, where $e = \text{Ex}(d,k)$
      and $d \in \text{St-DL}$. Then f is in α (1)

(15) Let $\Delta\varphi$ be the shortest prefix of Ω such that f is in $\omega(S,\Delta\varphi)$. Then

φ is a firing of d, and every entry which is in γ but not in

ω(S,Δ) is an input entry to e   Alg. 4.3-1

(16) There are at most In(l(d)) input entries to e in β, so there are

fewer than In(l(d)) in γ   (14)+(13)+Def. 4.2-6

(17) No entry which is in γ but not in ω(S,Δ) is an initiating entry

(16)+(15)+Def. 4.2-6

(18) Φ(γ) = Φ(ω(S,Δ)), which is the reduction of Δ   (17)+(11)+Def. 4.3-4

(19) Δ is the prefix of Ω whose reduction is Φ(γ)   (18)+(15)

(20) d is enabled in S·Δ   (15)+Def. 2.3-1

(21) Let γf be any prefix of β in which f = Ent(Ex(d,k),j) where d∈DL

and d = (c,n). Let b be the number-n program output arc of P, if

c = "OD", or else the number-n input arc of c. Then f is not in

α and there is a token on b in S·Ω   (1)+Alg. 4.3-1

(22) α is a prefix of γ, and every entry which is in γ but not in α is an

input entry to an execution Ex(d',k') where d'∈DL   (21)+Alg. 4.3-1

(23) Φ(γ) = Φ(α)   (22)+Defs. 4.2-6+4.3-4

(24) Ω is the prefix of Ω whose reduction is Φ(γ)   (23)+(12)

(25) There is a token on b in S·Ω, and since Ω is halted, there is no

firing sequence starting in S·Ω   (21)+Def. 2.3-1

(26) β is in $J_{S,\Omega}$ (1)+(12)+(13)+(14)+(19)+(20)+(21)+(24)+(25)+Def. 4.3-5

△

By the construction in Algorithm 4.3-1, the integer k in an execution

Ex(d,k) serves as an index among all the executions of d initiated in a

canonical computation. I.e., Ex(d,k_1) is initiated before Ex(d,k_2) only

if $k_1 < k_2$. The final general result presented here is that this indexing

property is exhibited by all computations. An intermediate deduction in

the proof will be needed directly in Chapter 7; for convenience, it is
here separated out and proven first.

Theorem 4.3-2 Let P be any data-flow program, and let Int(P) be (St, I, IE).
For any initial standard or modified state S for P and any halted firing
sequence Ω starting in S, let α be any prefix of any causal permutation of
$\eta(S,\Omega)$. Let θ be any prefix of Ω whose reduction is $\Phi(\alpha)$. Then for any
execution $e = Ex(d,k)$ where $d \in St-DL$, e is initiated in $\alpha \Rightarrow$ there are at
least k firings of d in θ.

Proof:

(1) $d \in St-DL \Rightarrow$ d is the label of an actor in P, and $I(d)$ is its action

Def. 4.3-2

(2) $\Rightarrow In(I(d)) > 0$ (1)+Defs. 4.3-1+2.1-5+2.1-2

Prove $\Rightarrow$ first, by induction on the length of α. All initiations and
reconstructions Φ are with respect to Int(P).

Basis: $|\alpha| = 0$.

(3) For any execution $e = Ex(d,k)$ where $d \in St-DL$, e is not initiated in

     α, so $\Rightarrow$ is vacuously true   (2)+Def. 4.2-6

Induction step: Assume $\Rightarrow$ is true for any α of length n≥0, and consider
$\alpha = \gamma f$ of length n+1, which is a prefix of some causal permutation β of
$\omega = \eta(S,\Omega)$.

(4) $\Phi(\gamma)$ is a prefix of $\Phi(\gamma f)$ Def. 4.3-4

(5) For any execution $e = Ex(d,k)$ where $d \in St-DL$, e is initiated in γ

     $\Rightarrow$ there are at least k firings of d in any prefix of Ω whose

     reduction is $\Phi(\gamma)$ ind. hyp.

(6) ⇒ there are at least k firings of d in any prefix of Ω whose

reduction is $\Phi(\gamma f)$ (4)+Def. 2.4-5

(7) f is not the initiating entry to an execution Ex(d,k) where d$\in$St-DL

⇒ [for every such execution e, e is initiated in γf ⇒ e is initiated

in γ ⇒ there are at least k firings of d in any prefix of Ω whose

reduction is $\Phi(\gamma f)$]   (5)+(6)+Def. 4.2-6

(8) Assume f is the initiating entry in γf of execution e = Ex(d,k)

where d$\in$St-DL. ω is a computation for Int(P)   Lemma 4.3-2

(9) ω has at most In(I(d)) input entries to e   (8)+Def. 4.2-6

(10) α and ω contain the same set of In(I(d))$\geqslant$0 input entries to e

           (9)+(8)+Def. 4.2-6

(11) There is some j such that b, the number-j input arc of d, has a

token removed at each firing of d   (1)+Def. 2.1-5

(12) Since there is at least one entry to e in ω, there are k firings

of d in Ω (10)+Alg. 4.3-1

(13) There is an entry g in ω, hence in α, whose transfer has destination

Dst(e,j)   (12)+(11)+(10)+Alg. 4.3-1

(14) Let Src(Ex(d',k'),i) be the source in T(g). Then there is a prefix

$\Delta\varphi$ of Ω in which φ is the kth firing of d, and φ removes a token

from b. d'$\in$DL ⇒ d'$\in$\{"IT","IF","ID"\} ⇒ the token removed from

b by φ is on b in S (11)+Alg. 4.3-1

(15) ⇒ φ is the first firing of d, so k = 1   (11)

(16) ⇒ since at least e is initiated in α, $\Phi(\alpha)$ has at least k firings

of d   Def. 4.3-4

(17) ⇒ any prefix of Ω whose reduction is $\Phi(\alpha)$ has at least k firings

of d   !   Def. 2.4-5

(18) Assume $d \in$ St-DL and $k > 1$. Then there are exactly k' firings of d'

in Δ (14)+(15)+Alg. 4.3-1

(19) Since g is an output entry of $Ex(d',k')$, the initiating entry of

$Ex(d',k')$ strictly precedes g in α, hence is in γ

                              (8)+(13)+(14)+Defs. 4.2-5+4.2-7

(20) $\Phi(\gamma f)$ is $\Phi(\gamma)\varphi$, where φ is a firing of d   (8)+Def. 4.3-4

(21) Let $\theta\varphi$ be any prefix of Ω whose reduction is $\Phi(\gamma f)$. Then the last

firing φ in $\theta\varphi$ is of d, and the reduction of θ is $\Phi(\gamma)$

                                     (20)+Def. 2.4-5

(22) There are at least k' firings of d' in θ (19)+(21)+(5)+(6)

(23) The $k-1^{st}$ firing of d removes a token from b, as does the kth

                                        (11)+(18)

(24) There is a prefix $\Xi\varphi'$ of Δ containing $k-1$ firings of d such that a

token appears on b in the transition from $S \cdot \Xi$ to $S \cdot \Xi\varphi'$ (23)+(14)

(25) Either φ' is a firing of d', or d' is a Select which is in a pool

in $S \cdot \Xi$ but is not in a pool in $S \cdot \Xi\varphi'$ (24)+Def. 3.3-9

(26) Let $\chi\varphi''$ be the prefix of Ω in which φ'' is the $k-1^{st}$ firing of d.

There is no firing of d' in $\chi \Rightarrow d'$ is not in a pool in $S \cdot \chi$

                                        (23)+Def. 3.3-9

(27) There is a firing of d' in $\chi \Rightarrow$ there is a longest prefix $\Psi\varphi_{d'}$ of χ

such that there is no token on b in $S \cdot \Psi$, but there is one in $S \cdot \Psi\varphi_{d'}$

$\Rightarrow d'$ is not in a pool in $S \cdot \Psi\varphi_{d'}$ (26)+(23)+Def. 3.3-9

(28) $\Rightarrow$ For every prefix Ψ' of Ω with $|\Psi\varphi_{d'}| \leq |\Psi'| \leq |\chi|$, there is a

token on b in $S \cdot \Psi'$, so d' is not enabled   (23)+Defs. 3.3-6+2.1-5

(29) $\Rightarrow$ d' is not in a pool in $S \cdot X$ (27)+Def. 3.3-9

(30) d' is not in a pool in $S \cdot X$ (26)+(27)+(29)

(31) d' is in a pool in $S \cdot \Xi \Rightarrow$ there is a firing of d' in Ξ, hence in Δ,

     but not in X at which d' is placed in a pool   (30)+Def. 3.3-9

(32) There is a firing of d' between the $k-1^{st}$ and k^{th} firings of d

                                             (24)+(25)+(31)

(33) Exactly k' firings of d' precede the kth firing of d, so at most

     $k'-1$ firings of d' precede the $k-1^{st}$ firing of d   (14)+(32)

(34) There are fewer than k firings of d in $\theta\varphi \Rightarrow$ the last firing φ is the

     n^{th} firing of d, for $n<k$ (21)

(35) $\Rightarrow$ At most $k'-1$ firings of d' are in θ (33)

(36) There are at least k firings of d in $\theta\varphi$ (34)+(35)+(22)

Thus it is proven by induction that

(37) $e = Ex(d,k)$ is initiated in $\alpha \Rightarrow$ there are at least k firings of d

     in θ (14)+(15)+(17)+(36)

Next prove the converse, by contradiction. Assume

(38) There is an α and an prefix θ of Ω whose reduction is $\Phi(\alpha)$, and some

     $d \in St-DL$ and $k>0$ such that there are at least k firings of d in θ,

     but $Ex(d,k)$ is not initiated in α

(39) Let $n \geq k$ be the number of firings of d in θ. Then there are n

     executions of d initiated in α (38)+Def. 4.3-4

(40) Since $Ex(d,k)$ is not initiated in α, there is an $m>n$ such that

     $Ex(d,m)$ is initiated in α (39)

(41) There are at least $m>n$ firings of d in θ (37)+(40)

Since (38) leads to a contradiction between (39) and (41), (38) is false

i.e., there are at least k firings of d∈St-DL in θ ⇒ Ex(d,k) is initiated in α.

△

Corollary 4.3-1 Let P be any data flow program and let Int(P) be (St,I,IE). Let S be any initial standard or modified state for P, and let Ω be any halted firing sequence starting in S. Let β be any computation in $J_{S,\Omega}$. For any d∈St-DL and integer k such that Ex(d,k) is initiated wrt Int(P) in β, the initiating entry to that execution is preceded in β by exactly k-1 initiating entries to other executions of d.

Proof: By induction on n, the number of executions of d initiated in each prefix of β (all initiations and reconstructions are wrt Int(P)).

(1) In(I(d)) > 0   Defs. 4.3-2+4.3-1+2.1-5+2.1-2

(2) For any prefix α of β, let θ be any prefix of Ω whose reduction is
  Φ(α). Then the number of firings of d in θ equals the number of
  executions of d initiated in α   Defs. 4.3-4+2.4-5

Basis: n = 1.

(3) There is one firing of d in θ   (2)

(4) Let Ex(d,k) be the one execution of d initiated in α. There are
  at least k firings of d in θ   (2)+Thm. 4.3-2

(5) k = 1   (3)+(4)

Induction step: Assume the Corollary is true for the first n initiating entries to executions of d in β, n>0.

(6) Let γ be the shortest prefix of β in which there are exactly n
  executions of d initiated. Then any initiating entry in γ to an
  execution Ex(d,i) is preceded by the initiating entries to exactly
  i-1 other executions of d   ind. hyp

MICROCOPY RESOLUTION TEST CHART
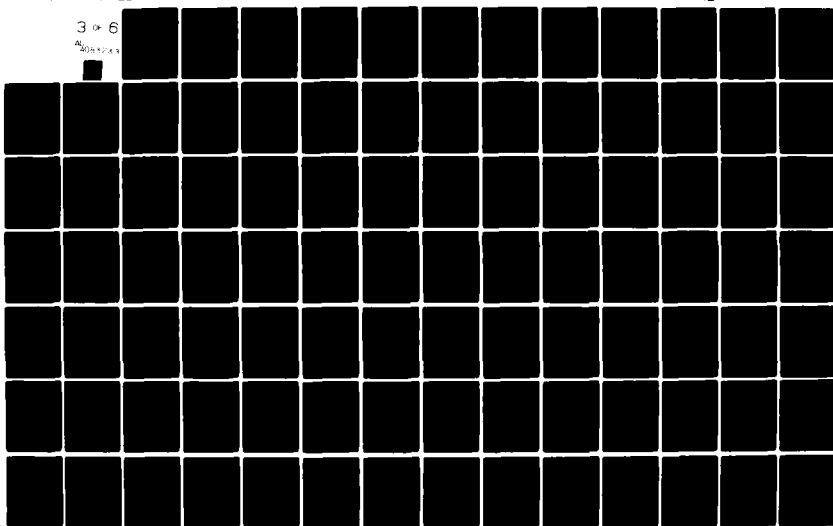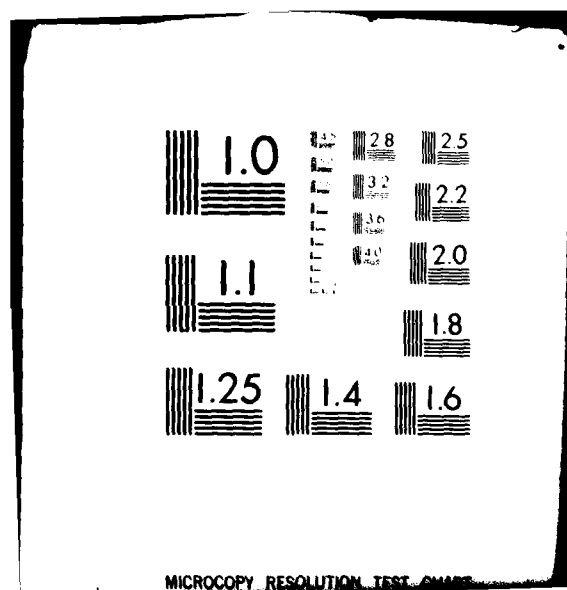
(7) Since there are only n initiating entries in γ, the initiating entry
    to any execution of d in γ is preceded by the initiating entries
    to at most n-1 other executions of d   (1)+(6)

(8) Ex(d,i) is initiated in $\gamma \Rightarrow i-1 \leq n-1 \Rightarrow i \leq n$ (6)+(7)

(9) Since there are n executions of d initiated in γ, Ex(d,i) is
    initiated in γ iff $i \leq n$ (6)+(8)

(10) Let Ex(d,k) be the execution of d whose initiating entry is $n+1^{st}$
     in β; i.e., that entry is preceded in β by exactly n other init-
     iating entries to executions of d. Let α be the shortest prefix
     of β containing that initiating entry. Then there are exactly
     n+1 firings of d in θ (2)

(11) There are at least k firings of d in θ, so $k \leq n+1$ (10)+Thm. 4.3-2

(12) Ex(d,k) is not initiated in γ (6)+(10)+(1)

(13) $k < n+1 \Rightarrow$ Ex(d,k) is initiated in γ (9)

(14) $k = n+1$ (11)+(13)+(12)

(15) The $n+1^{st}$ execution of d initiated in β is Ex(d,n+1)   (10)+(14)

(16) For any $k \leq n+1$, if Ex(d,k) is initiated in β, then its initiating
     entry is preceded in β by the initiating entries to exactly k-1
     other executions of d   (9)+(6)+(15)

$\triangle$

Chapter 5

Structure-as-Storage Models

This thesis is concerned with defining structure operations which make it easy to prove that parallel programs using them are functional. The entry-execution model has been introduced as the vehicle for a general proof of determinacy (which implies functionality). Thus the definitions of the structure operations must be drawn within the framework of this model. This Chapter uses the operations in the data-flow language L_{BS} to illustrate a suitable mode of definition.

Constraints on computations are used to "define" structure operations in the following sense: This Chapter gives a set of example constraints. Any model satisfying all of these is called a Structure-as-Storage (S-S) model. These constraints concern only the input and output entries of executions of structure operations, and are constructed so that $EE(L_{BS},S)$ is an S-S model (this latter point is proven in Section 5.3 below). Therefore, any other language whose model is S-S may be considered to have the same set of structure operations as L_{BS}.

The L_{BS} structure operations have already been defined using a schema model of data flow (Definition 2.2-5). According to this definition, a structure operation selected to fire in a state S outputs values depending on just its inputs at that firing and the heap in S. In the entry-execution model, even though the concept of state has been abstracted away, it is not difficult to define the heap determined by a computation; this is in fact done in Section 5.2. The output entries of an execution e could then be constrained to depend just on e's input entries and the heap

determined by the computation immediately preceding e's initiation. To do
so, however, would greatly encumber a constructive proof along the lines
of the Determinacy Proof Technique: It would be necessary at each step
to characterize not only the newly-constructed computation, but also the
heap it determines.

A computation alone contains enough information to ascertain the
correct output of most executions. It has already been decided to employ
computations to convey both the order in which executions occur and the
values of their inputs and outputs. It is here further decided to fore-
sake the heap concept, in favor of using computations to express directly
the complex interrelationships which exist among executions of structure
operations.

5.1 The Constraints

The constraints defining a Structure-as-Storage model are listed
next. The remainder of this section presents the constraints and
describes how each is derived from the heap-oriented model of L_{BS}.

__Definition 5.1-1__ An entry-execution model (V, L, A, In, E) is a
__Structure-as-Storage__ (S-S) model iff:

1. There is a distinct subset V_p of the atomic value domain V. Atomic
   values in V_p are __of__ __pointer__ __type__. All other values in V are __of__
   __non-pointer__ __type__; included among these are __nil__ and __undef__.

2. The domain A includes the following eight specific actions, and In
   assigns to each the indicated input arities: Fetch (1), First (1),
   Next (2), Select (2), Copy (1), Assign (2), Update (3), and
   Delete (3).

3. For every expansion (Int,*J*) in *E*, for every job J∈*J*, every computa-
    tion in J satisfies the following constraints (all given later):

    The Input/Output Type Constraint

    The Atomic Output Constraint

    The Structure Output Constraint

    The Unique Pointer Generation Constraint

Furthermore, every pair of computations in J satisfies the Initial
Structure and First/Next Output Constraints, and J itself satisfies the
Pointer Transparency Constraint.

△

5.1.1 Input/Output Types

From Definition 2.2-5, it is apparent that certain inputs to structure
operations must be of pointer type. Furthermore, it is generally true that
no other input to these or to other operations may be of pointer type; for
example, it is not possible to do arithmetic on pointers. The exception
to this rule are the pI actions:

Definition 5.1-2 Given a model (V, L, A, In, E), an action a∈*A* is a
pseudo-identity (pI) action iff the following is true for every interpre-
tation Int = (St,*I*,IE) in the model: For any execution e of a and
computation ω for Int:

1. There is no entry in ω whose transfer has source Src(e,i) for i≠1.

2. There is a j, depending only on the non-pointer-valued input
    entries to e in ω, such that the value of Src(e,1) in ω (if any)
    equals the value of $\text{Ent}_\omega(e,j)$, the number-j input entry to e in ω.

△

The constraints on the types of the inputs and outputs of execution e are translated from Definition 2.2-5 into:

<u>Constraint 5.1-1</u> A computation ω for interpretation (St, I, IE) satisfies the <u>Input/Output</u> <u>Type</u> <u>Constraint</u> iff, for all d∈St and k>0, the input and output entries of Ex(d,k) in ω are as described below, depending on I(d).

Structure operations - Table 5.1-1 gives the type of input entry and source values of each execution e of a structure operation.

pI actions - The values of any input or output entries of an execution of a pI action or of an execution in IE may be either of pointer or of non-pointer type.

All others - The value of any input or output entry of an execution of any other action must be of non-pointer type.

$\triangle$

5.1.2 Pointer Transparency

A pointer's only identity stems from its uniqueness; i.e., the only relationship possible between two pointers is that they are distinct.

Operation	Input Entries			Sources	
	Ent(e,1)	Ent(e,2)	Ent(e,3)	Src(e,1)	Src(e,2)
Fetch	ptr	---	---	non-ptr	non-ptr
First	ptr	---	---	non-ptr	non-ptr
Next	ptr	non-ptr	---	non-ptr	non-ptr
Select	ptr	non-ptr	---	ptr	non-ptr
Copy	ptr	---	---	ptr	ptr
Assign	ptr	non-ptr	---	non-ptr	non-ptr
Update	ptr	non-ptr	ptr	non-ptr	non-ptr
Delete	ptr	non-ptr	---	non-ptr	non-ptr

Input/Output Types

Table 5.1-1

Any set of n distinct pointers will serve in any capacity (such as dom Π in a heap) equally as well as any other set of n distinct pointers. This gives rise in the S-S entry-execution model to the principle of pointer transparency: For any computation ω, let $\{p_1, p_2, \ldots, p_n\}$ be the set of distinct pointers appearing as the values of entries in ω, and let $\{q_1, q_2, \ldots, q_n\}$ be any other same-size set of distinct pointers. Replacing p_i wherever it appears as the value of an entry in ω with q_i, for $i = 1, 2, \ldots, n$, yields a new computation which is identical to within pointer values to ω. Any computation so related to ω is in every job that ω is in.

Definition 5.1-3 Two computations ω_1 and ω_2 are identical to within pointer values, written

$$\omega_2 \simeq \omega_1$$

iff there is a total one-to-one mapping Y over V_p such that ω_2 can be derived from ω_1 by substituting for each entry $f \in \omega_1$ a similar entry, whose transfer is T(f) and whose value is given by

if V(f) is not a pointer, then V(f), else Y(V(f)).

$\triangle$

Constraint 5.1-2 A job J satisfies the Pointer Transparency Constraint iff for any computation $\omega_1 \in J$ and any other computation ω_2,

$$\omega_2 \simeq \omega_1 \Rightarrow \omega_2 \in J$$

$\triangle$

5.1.3 The Concept of Reach

As mentioned earlier, the intent here is to define the L_{BS} structure operations without reference to a heap. That is, the outputs of any execution of such an operation are to depend upon just the inputs to that and

to previously-initiated executions. The current sub-section analyzes this dependence precisely, using the schema model of L_{BS} from Chapter 2; the algorithm of Section 4.3 will then be used to take this dependence into constraints on computations in $EE(L_{BS}, S)$.

The most important new concept in a heapless definition of structure operations is that of the <u>reach</u> of a write-class firing (Assign, Update, or Delete.) This section concentrates on defining the reach $R(A)$ of an Assign firing A in a firing sequence Ω; the reach of an Update or Delete is a straightforward extension of this. The principle of reach is that $R(A)$ should consist of just each firing φ in Ω for which the state change effected by φ depends directly on the atomic input to A. The significance of this principle is two-fold: 1) The output of a Fetch firing F is necessarily equal to the atomic input of that unique Assign firing into whose reach F falls. 2) For any Assign operator d in a determinate program, the reach of the kth firings of d in all firing sequences starting in equal initial states for the program is the same.

Two other new concepts — access history and duration — must be introduced before reach can be defined. The access history for pointer p in firing sequence Ω, H_p, is the sub-sequence of structure operator firings in Ω whose pointer inputs equal p; it may be said that these are the firings in Ω which "access the node $n = \Pi(p)$." For any Assign firing A in Ω which accesses n, the duration $D(A)$ is defined as follows: Let v be the value of A's atomic input. Then $D(A)$ is the set of firings in Ω which access either n or a copy of n during a period when that node's value must equal v.

The reach of A is a subset of the duration of A. Precise identification of all the firings in D(A), and of that subset of these which is R(A), is done in two steps: first all those firings which access n, and then those which access copies of n.

1. Firings which access n -

Let A' be the next Assign firing following A in H_p, if any. Then D(A) contains (among others) all firings in H_p following A but not following A' (this includes A'). The effect of firing A is to assign the value in SM(n) to be v, and the effect of A' is to change that value. Therefore, the value of SM(n) is guaranteed to be v just at those firings after A but not after A'. I.e., each firing in D(A) which accesses node n does so while the value in SM(n) is guaranteed to be v.

A firing F of a Fetch operator d in D(A) effects a state change which is distinguished by the placement of tokens with value v on d's output arcs. This dependence means that every Fetch firing in the duration of A is in the reach of A. Similarly, the control output of A' depends on whether or not the value in SM(n) just before that firing is nil. Since this value is the one assigned by A, A' (if it exists) is in R(A).

Finally, let C be any Copy firing in D(A) which accesses n. This activates a new node m. SM(m) in the state immediately following C equals SM(n) in the state immediately preceding C. Since C is in D(A), the value in that content equals the value assigned by A. Therefore, any Copy firing in D(A) is also in R(A). Copy, Fetch, and Assign operators are the only ones which effect state changes that depend on the value

assigned to a node; hence the reach of an Assign firing contains only Fetch, Assign, and Copy firings.

2. Firings which access copies of n –

Let C be any Copy firing in D(A) which accesses n, let q be its pointer output, and let $m = \Pi(q)$. Then the initial value in SM(m) equals v. Define the initial-value duration for q, D_q, to be the set of all firings in H_q not following the first Assign firing (if any). Then D_q consists of just those firings which access m while it still has its initial value v. Therefore, for any firing φ in D_q of a Fetch, Assign, or Copy, the state change effected depends directly on the atomic input to A, and so φ belongs in the reach of A. By defining D(A) to include D_q, R(A) remains just all Fetch, Assign, or Copy firings in D(A).

The above paragraph is true for the pointer output of any Copy firing which accesses m in D_q. In general, this reasoning can be applied recursively to yield: D(A) consists of all those firings accessing n which are identified in (1) above, plus the initial-value duration for any pointer output of any Copy firing in D(A). R(A) then consists of all Fetch, Assign, and Copy firings in D(A).

It can now be seen how this precise development of the concept of reach leads to constraints on the outputs of Fetch and Assign firings: The access histories in a firing sequence are all disjoint. An access history is partitioned by the set of durations consisting of its initial-value duration and the duration of each Assign firing appearing in it. Therefore, the durations of Assign firings are all disjoint, and so each Fetch or Assign firing F is in at most one reach. If F is in R(A) for

Assign firing A, then its data output must equal v, the atomic value input

to A (if F is a Fetch firing; the data outputs of Assign firings are

identically zero). The control outputs of F equal the value of the pred-

icate "v is not nil". (The case that F is not in the reach of any Assign

firing is examined later.)

The concepts of access history, duration, and reach can all be

defined for an entry-execution model by applying Algorithm 4.3-1 to the

descriptions just given. This is done in the next sub-section. The

concepts are extended to the Select, Update, and Delete operations in

Section 5.1.5.

5.1.4 The Atomic Output Constraint

Definition 5.1-4 Given any interpretation Int and a computation ω for Int,

the access history H_p^ω for any pointer p in ω is a sequence of all the

entries in ω whose values equal p and whose target executions are initiated

in ω (with respect to Int). In this sequence, $Ent(e_1, j_1)$ follows

$Ent(e_2, j_2)$ iff e_1's initiating entry follows e_2's in ω.

$\triangle$

In the just-concluded schema-model development of the concept of reach,

an access history was defined as a sequence of firings. The entry-

execution analog of a firing is an execution. Thus it might be expected

that an access history would be defined here as a sequence of executions.

The reason for using a sequence of entries instead is that each Update

execution U has two pointer-valued input entries. It is necessary that

each appearance of U in an access history H_p^ω be qualified as to which of

its input entries has value p. Since this information is inherent in the entries of ω, the form of access history defined above is more convenient. It should be noted that the entries appear in the order of initiation of their target executions, which is not necessarily the same as their order of appearance in ω.

Access histories have a significance of their own beyond their role in defining reach: Let ω_1 and ω_2 be any two computations in a job from a determinate expansion. Then for every access history in ω_1, there is an access history in ω_2 containing exactly the same entries.

Definition 5.1-5 Let ω be any computation for any interpretation, and denote by AS the set of Assign executions initiated in ω. For each $A \in AS$, the duration of A, $D(A)$, in ω is a set of entries in ω defined recursively as follows: Let $APS = \{Ent(e,1) \mid e \in AS\}$.

(1) Let H be the access history containing $f = Ent(A,1)$. Then the set

$\{g \mid g$ follows f in H with no intervening entry from APS$\}$

is contained in $D(A)$.

(2) Let C be any Copy execution such that $Ent(C,1) \in D(A)$, and let q be its pointer output. Then the set

$\{g \mid$ no entry from APS precedes g in $H_q^\omega\}$

is contained in $D(A)$.

(3) $D(A)$ consists of just those entries derived from (1) and (2) above.

$\triangle$

If $Ent(e,1)$ follows $Ent(A,1)$ in an access history and no entry from APS appears between them, then:

1. A and e both have the same pointer input,

2. e initiates after A, and

3. no Assign execution A' with the same pointer input initiates between

    A and e

If ω is from the model $EE(L_{BS},S)$, then ω models a firing sequence Ω, and

A and e model two firings. One of these firings is of an Assign, and

the other appears after it, but not after the next Assign firing, in the

same access history. Therefore, the execution e models a firing which

belongs in the duration of the firing modeled by execution A.

Definition 5.1-6 Let ω be any computation for any interpretation. For

each Assign execution A initiated in ω, the reach of A, R(A), in ω is the

set of executions consisting of each Fetch, Assign, or Copy execution e

for which Ent(e,1) is in D(A).

$\triangle$

The outputs of all Fetch and Assign executions in R(A) depend just on A's

atomic input, as detailed in the following:

Constraint 5.1-3 A computation ω for any interpretation satisfies the

Atomic Output Constraint iff, for every Fetch or Assign execution e which

is in the reach of some Assign execution A in ω, the values of Src(e,1)

and Src(e,2) are as follows, where v = V(Ent(A,2)).

Action of e	value of Src(e,1) (Data outputs)	value of Src(e,2) (Control outputs)
Fetch	v	v $\neq$ nil
Assign	0	v $\neq$ nil

$\triangle$

5.1.5 The Structure Output Constraint

The only other operations whose firings can affect other firings' outputs are Update and Delete. This effect could be made precise by defining the reach of such a firing U in a schema model of L_{BS}. There would be two differences between this and the earlier description of the reach of an Assign firing, both of which are due to the fact that U affects just one branch in the content of the node it accesses:

1. U's duration D(U) is ended only by another Update or Delete firing accessing the same node and having the same selector input.

2. U affects only those firings in D(U) which depend upon the existence of the branch which U changes. Therefore, the reach of U, R(U), contains firings only of:

   a. a Select with the same selector input as U,

   b. an Update or Delete with the same selector input as U (because control outputs are affected), or

   c. a Copy, First, or Next.

The state change effected by a firing φ of a Copy, First, or Next which accesses a node n depends on the entire set of branches in SM(n). This in turn depends on every Update or Delete firing into whose duration φ falls. Therefore, any such firing in D(U) is in R(U).

Since the reach of an Update or Delete is so similar to that of an Assign, the step of defining it for a schema model will be bypassed. Instead, it is defined directly for an entry-execution model:

__Definition 5.1-7__ Let ω be any computation for any interpretation, and denote by SS the set of all Update and Delete executions initiated in ω. Then for each $U \in SS$, the __duration__ of U, D(U), in ω is defined recursively as follows. Let

$$SPS(U) = \{Ent(e,1) \mid e \in SS \text{ and } V(Ent(e,2)) = V(Ent(U,2))\}.$$

(1) Let H be the access history containing $f = Ent(U,1)$. Then the set

$$\{g \mid g \text{ follows } f \text{ in H with no intervening entry in } SPS(U)\}$$

is in D(U).

(2) Let C be any Copy execution such that Ent(C,1) is in D(U), and let q be its pointer output value. Then the set

$$\{g \mid \text{no entry in } SPS(U) \text{ precedes g in } H_q^\omega\}$$

is in D(U).

(3) D(U) consists of just those entries derived from (1) and (2) above.

$\triangle$

SPS(U) contains just those entries which, by virtue of their targets having the same selector input, could end U's duration.

__Definition 5.1-8__ Let ω be any computation for any interpretation. For each Update or Delete execution U initiated in ω, the __reach__ of U, R(U), in ω is the set of executions

$$\{e \mid Ent(e,1) \in D(U), e \text{ is an execution of a Select, Update, or}$$

$$\text{Delete, and } V(Ent(e,2)) = V(Ent(U,2))\}$$

$$\cup \{e \mid Ent(e,1) \in D(U) \text{ and e is an execution of a Copy, First, or Next}\}.$$

$\triangle$

The exact dependence on U of the output of a First or Next execution in R(U) is a complex issue, and will be considered later. The dependence on

U of the outputs of a Select, Update, or Delete execution in R(U) is easily understood:

Constraint 5.1-4 A computation ω for any interpretation satisfies the Structure Output Constraint iff, for every Select, Update, or Delete execution e which is in the reach of some Update or Delete execution U in ω, the values of Src(e,1) and Src(e,2) in ω are as follows, depending on the actions of e and U:

U is an Update -

Action of e	value of Src(e,1) (Data outputs)	value of Src(e,2) (Control outputs)
Select	V(Ent(U,3))	true
Update or Delete	0	true

U is a Delete -

Action of e	value of Src(e,1) (Data outputs)	value of Src(e,2) (Control outputs)
Select	undef	false
Update or Delete	0	false

$\triangle$

5.1.6 Initial Structures

Section 5.1.3 develops the concept of reach and uses it to relate the output of a Fetch firing F to the input of the Assign firing into whose reach F falls. This current sub-section is concerned with finding the output of a Fetch firing which is not in any reach (a Fetch cannot be in the reach of anything but an Assign). As before, answers are developed in

the standard schema model first, and then carried over into the entry-execution model.

Let S be any initial state and Ω any firing sequence starting in S. Then a Fetch firing which is not in any reach in Ω necessarily outputs the value stored at some node in S. The identity of this node is determined with the aid of the concept of dynamic descendancy, defined thus: Node n is dynamically descended from node m in Ω iff:

1. n = m, or

2. n is activated at some Copy firing in Ω which accesses a node
    which is dynamically descended from m in Ω.

Every node in $S \cdot \Omega$ either is in S or is activated by a unique Copy firing in Ω, which accesses some existing node. Therefore, for any such node n, there is a node m in S from which n is dynamically descended in Ω. If a Fetch firing F which accesses node n is not in any reach, it will output the value v in SM(m) in S, as the following argument shows:

Let q be the pointer to n; i.e., $\Pi(q) = n$. Then F is in the access history H_q in Ω, and is not therein preceded by any Assign firing (for then F would be in the reach of that firing). Therefore no new value is stored at n before F, so F outputs the initial value of n. If n = m, then that value is v. If $n \neq m$, prove by induction on the length of its dynamic descendancy that its initial value is v.

Since n is not in S, it is activated by some Copy firing C. The node n' accessed by C is dynamically descended from m. The initial value of n is the value of n' in the state in which C fires. Since F is not in the reach of any Assign, C is not either. Therefore, no firing preceding C changes the value of n', so in the state in which C fires,

n' has its initial value. I.e., the initial value of n equals the initial value of n', which by induction hypothesis equals the initial value of m.

In a schema model, then, the output of any Fetch firing not in a reach is determined to be equal to a value stored in the initial state. This result cannot be carried over to the entry-execution model, however, because all concept of state has been abstracted away. Fortunately, there is a weaker conclusion which can be expressed in the entry-execution model and is sufficient for the purpose of proving determinacy:

Consider two different firing sequences Ω_1 and Ω_2 starting in initial states S_1 and S_2 such that S_2 equals S_1 under some mapping I. Suppose that Fetch firings F_1 and F_2 access nodes n_1 and n_2 in Ω_1 and Ω_2 respectively, and that m_1 (m_2) is the node in S_1 (S_2) from which n_1 (n_2) is dynamically descended. If $m_2 = I(m_1)$ then in the initial states, $SM(m_2) = I(SM(m_1))$, implying that m_1 and m_2 have the same initial value v. Thus if neither F_1 nor F_2 falls into the reach of any Assign firing, both will output v.

This observation can be used to constrain certain Fetch executions to have the same outputs in two computations in the same job (which is all that is required for determinacy). The constraint is developed in several steps: First it is noted that every such pair of computations ω_1 and ω_2 is derived from a pair of firing sequences Ω_1 and Ω_2 starting in two initial states S_1 and S_2. Since ω_1 and ω_2 are in the same job, S_2 must equal S_1 under some mapping I.

The second step is to identify directly from ω_1 (or ω_2) which pairs of pointers p and q are related thusly: the node pointed to by p is

dynamically descended in Ω_1 (or Ω_2) from the node pointed to by q. This is easily done, as follows:

Definition 5.1-9 Let ω be any computation for any interpretation, and let p be any pointer. Then p is <u>dynamically descended in</u> ω from a pointer q, written $DD_\omega(q,p)$, iff either

1. p = q, or

2. p is the value in ω of an output entry of a Copy execution the value of whose input entry is a pointer dynamically descended from q in ω.

$\triangle$

Let the heap in the initial state S_i, i=1,2, be (N_i, Π_i, SM_i). Then, for each pointer p appearing as the value of an entry in ω_i, there is a unique pointer q in dom Π_i from which p is dynamically descended (Lemma 5.2-4 below).

The third and final step in developing the constraint is to define a relation ρ over pointer-computation pairs. Two such pairs should be related by ρ, written $(p_1, \omega_1) \rho (p_2, \omega_2)$, iff, for q_1 and q_2 the pointers in dom Π_1 and dom Π_2 from which p_1 and p_2 are dynamically descended in ω_1 and ω_2, $\Pi_2(q_2) = I(\Pi_1(q_1))$; then, it has been argued, any two Fetch executions with p_1 and p_2 as inputs in ω_1 and ω_2 are constrained to have equal outputs, if neither falls into a reach. The relation ρ is first recursively derived for p_1 and p_2 which are themselves in dom Π_1 and dom Π_2.

The basis is that if p_1 and p_2 are on the same program input arc in S_1 and S_2, respectively, then $\Pi_2(p_2) = I(\Pi_1(p_1))$ (by definition of equal states), so $(p_1, \omega_1) \rho (p_2, \omega_2)$. The induction step involves Select executions S_1 and S_2 initiated in ω_1 and ω_2 with the same selector input s and

pointer inputs p_1 and p_2. For $i=1,2$, if S_i does not fall into a reach in ω_i, then its pointer output is p_i', where $(s,\Pi_i(p_i'))\in SM_i(\Pi_i(q_i))$, q_i being the unique pointer in dom Π_i such that $DD_{\omega_i}(q_i,p_i)$. (This is by an argument analogous to that given earlier concerning the output of a Fetch firing which is in no reach.) Then this series of inferences can be drawn: $(p_1,\omega_1)\rho(p_2,\omega_2) \Rightarrow \Pi_2(q_2) = I(\Pi_1(q_1)) \Rightarrow SM_2(\Pi_2(q_2)) = I(SM_1(\Pi_1(q_1)) \Rightarrow \Pi_2(p_2') = I(\Pi_1(p_1')) \Rightarrow (p_1',\omega_1)\rho(p_2',\omega_2)$, since p_1' is in dom Π_1.

Finally, for any two pointers $q_1 \in$ dom Π_1 and $q_2 \in$ dom Π_2, and two other pointers $p_1 \neq q_1$ and $p_2 \neq q_2$, $(q_1,\omega_1)\rho(q_2,\omega_2) \wedge DD_{\omega_i}(q_i,p_i) \Rightarrow \Pi_2(q_2) = I(\Pi_1(q_1))$ $\wedge$ q_i is the unique pointer in dom Π_i from which p_i is dynamically descended in $\omega_i \Rightarrow (p_1,\omega_1)\rho(p_2,\omega_2)$. The relation ρ is defined concisely next; a proof that $(p_1,\omega_1)\rho(p_2,\omega_2)$ iff $\Pi_2(q_2) = I(\Pi_1(q_1))$ may be found in a later section (Theorem 5.3-2).

Definition 5.1-10 Given any interpretation Int = (St,I,IE), the equal pointer relation is a binary relation over the set of all ordered pairs (p,ω) where p is a pointer and ω is a computation for Int. Two such pairs (p_1,ω_1) and (p_2,ω_2) are in this relation, written

$$(p_1,\omega_1)\rho(p_2,\omega_2)$$

iff one of the following three statements is true:

1. There is a source s = Src(e,i) for some $e \in IE$ and some i such that p_1 is the value of s in ω_1 and p_2 is the value of s in ω_2.

2. There are two Select executions S_1 and S_2 such that:

    for $i=1,2$, p_i is the value in ω_i of Src$(S_i,1)$,

    for $i=1,2$, S_i does not fall into a reach in ω_i,

    $V(Ent_{\omega_1}(S_1,2)) = V(Ent_{\omega_2}(S_2,2))$, and

$$(V(Ent_{\omega_1}(S_1,1)),\omega_1)\rho(V(Ent_{\omega_2}(S_2,1)),\omega_2)$$

3. There is a pointer $q \neq p_1$ such that $DD_{\omega_1}(q,p_1)$ and $(q,\omega_1)\rho(p_2,\omega_2)$.

$\triangle$

One constraint arises immediately from the claim that for any $p_1 \in dom\ \Pi_1$ and $p_2 \in dom\ \Pi_2$, $(p_1,\omega_1)\rho(p_2,\omega_2)$ iff $\Pi_2(p_2) = I(\Pi_1(p_1))$: since Π_1, Π_2, and I are all one-to-one, given ω_1, ω_2, and, say, p_1, there is at most one p_2 such that $(p_1,\omega_1)\rho(p_2,\omega_2)$. It is also possible now to state a specific circumstance under which a Fetch execution must have the same output in different computations in the same job. This is combined with the analogous constraint on Select execution outputs in the following:

<u>Constraint 5.1-5</u> Given an interpretation Int, any pair ω_1 and ω_2 of computations for Int satisfies the <u>Initial Structure Constraint</u> iff the following are all true, where ρ is the equal pointer relation defined from Int:

1. For i=1,2, let p_1 and p_{i+2} be any two pointers such that neither is the value of an output entry of a Copy execution in ω_i. If $(p_1,\omega_1)\rho(p_2,\omega_2)$ and $(p_3,\omega_1)\rho(p_2,\omega_2)$, then $p_3 = p_1$, and if $(p_1,\omega_1)\rho(p_2,\omega_2)$ and $(p_1,\omega_1)\rho(p_4,\omega_2)$, then $p_4 = p_2$.

2. Let e_1 and e_2 be any two Fetch or two Assign executions initiated in ω_1 and ω_2 respectively with pointer inputs p_1 and p_2 such that $(p_1,\omega_1)\rho(p_2,\omega_2)$. If neither falls into a reach, then for i=1,2, the values of $Src(e_1,i)$ in ω_1 and $Src(e_2,i)$ in ω_2 are the same.

3. Let e_1 and e_2 be any two Select, Update, or Delete executions initiated in ω_1 and ω_2 with equal selector inputs and pointer inputs p_1 and p_2 such that $(p_1,\omega_1)\rho(p_2,\omega_2)$. If neither is in a reach, then

a. the values of $Src(e_1,2)$ in ω_1 and $Src(e_2,2)$ in ω_2 are equal, and

b. if both are Update or Delete executions, then the values of

$Src(e_1,1)$ in ω_1 and $Src(e_2,1)$ in ω_2 are the same (e.g., zero).
$\triangle$

It will be noted that there is no way to relate the output values of, e.g. two Fetch executions when exactly one falls into a reach. Fortunately, the need never arises.

5.1.7 The First/Next Output Constraint

This constraint concerns the outputs of two First or Next executions in two computations in the same job. It is similar to the Initial Structure Constraint, and is here developed in the same manner: first for the schema model, then for the entry-execution model.

Let S be any initial state and let Ω be any firing sequence starting in S. Let FN be any firing in Ω of a First or a Next operator, and let n be the node accessed by FN. Then the outputs of FN depend just on the set of selectors in $SM(n)$ in the state in which FN fires. A selector s is in this set iff:

1. FN is in the reach of an Update (not a Delete) firing having the selector input s, or

2. FN is not in any such reach and s is in $SM(m)$ in S, where n is dynamically descended from m in Ω.

This leads to the following sufficient condition under which two First or Next executions output the same value in different computations in a job in $EE(L_{BS},S)$:

<u>Constraint 5.1-6</u> Given an interpretation Int, any pair ω_1 and ω_2 of computations for Int satisfies the <u>First</u>/<u>Next</u> <u>Output</u> <u>Constraint</u> iff the following is true, where ρ is the equal pointer relation defined from Int:

Let e_1 and e_2 be two First executions, or two Next executions with the same selector inputs, initiated in ω_1 and ω_2 respectively. Then for $i=1,2$, the values of $Src(e_1,i)$ in ω_1 and $Src(e_2,i)$ in ω_2 are the same if:

1. the values of the pointer inputs to e_1 and e_2 are p_1 and p_2 such that $(p_1,\omega_1)\rho(p_2,\omega_2)$, and

2. for each selector s, e_1 is in the reach of an Update (Delete) execution with selector input s in ω_1 iff e_2 is in the reach of an Update (Delete) execution with selector input s in ω_2.

$\triangle$

5.1.8 The Unique Pointer Generation Constraint

In a schema model of an S-S language, the constraint on the pointer output by a Copy firing is quite elementary: Letting (N,Π,SM) be the heap in the state in which the Copy fires, the pointer which it outputs must be distinct from all those in dom Π. The corresponding constraint in an entry-execution model is more complex, however, due to the absence of any concept of heap. The problem there may be stated as: Given a computation in which a Copy execution has output entries of value p, from which other entries' values must p be distinct. The solution is developed below for $EE(L_{BS},S)$.

Let (Int,J) be any expansion from $EE(L_{BS},S)$, where $Int = (St, I, IE)$. Let ω be any computation in any job in J; then ω is a computation for Int. Let S be the initial state, and Ω the firing sequence starting in S, such

that ω is a prefix of some computation in $J_{S,\Omega}$. Then for any pointer q
which appears as the value of entries in ω, the first such entry must be
an output entry of either an input execution (one in IE), a Copy execution,
or a Select execution which is in no reach: The only other executions
which can possibly have pointer-valued output entries are pI executions
and Select executions which are in reaches (Constraint 5.1-1). If a pI
execution e has an output entry of value q, then it has an input entry of
value q, which must precede that output entry in ω. If a Select execution
S is in the reach of an Update execution U, then S's output entries have
the same value as Ent(U,3). U is initiated before all executions in its
reach, so Ent(U,3) must precede all output entries of S.

Let (N,Π,SM) be the heap in the initial state S. If q is the value
of an output entry of an execution in IE, then there is a token with value
q in the configuration in S; thus q must be in dom Π. If q is the value
of an output entry of a Select execution S which is in no reach, let s be
S's selector input. By analogy with the argument given earlier for Fetch
executions, the pair (s,Π(q)) must be in the content of some node in N,
which implies also that q must be in dom Π. In either case, p, being the
output of a Copy execution, must be distinct from q. Finally, p clearly
must be distinct from the value of the output entries of any other Copy
execution in ω. These conclusions are summarized in:

Constraint 5.1-7 Given an interpretation Int = (St, I,IE), a computation
ω for Int satisfies the Unique Pointer Generation Constraint iff the
following is true: Let C be any Copy execution initiated in ω, and let

p the value of its output entries in ω (if any). Then p is not equal to
the value of the output entries of any execution which is:

1. in IE, or

2. a Copy execution other than C, or

3. a Select execution which is in no reach in ω.

△

This completes the definition of a Structure-as-Storage (S-S) model.
The system of constraints just presented illustrates a mode of specifying
sets of operations in the medium of the entry-execution model: all
languages whose models are S-S contain some common set of operations. In
particular, since it is claimed that these constraints were constructed so
that $EE(L_{BS},S)$ is an S-S model, the set of structure operations in L_{BS}
has now been formally described. Section 5.3 proves rigorously the
validity of this claim; first, Section 5.2 develops a new concept import-
ant not only to that proof, but also to an appreciation of the information
content of an entry-execution computation.

As noted earlier, the work of Greif [19] is closely related to the
entry-execution model and to the use of constraints on computations to
specify operations. She studied the behaviors of actor systems. A
behavior is a partial order of events, which are closely analogous to
entries. A given actor system with given initial conditions may exhibit
several different behaviors; similarly, in the entry-execution model, a
given program and input expands into a job, a set of sequences of entries.
The two models are best brought into correspondence by viewing a job as
the set of all total orders of entries compatible with all the possible
behaviors for a given system and initial conditions.

Various techniques for coordinating parallel processes were described by the additional orderings they imposed on all possible behaviors (i.e., orderings beyond those inherent in the individual processes). To quote the most relevant example, a single cell of read/write storage (analogous to a node) was defined by constraints on events, which depended on the order in which operations were performed on the cell. That is, for a given cell, each behavior had to specify some total ordering of all operations on that cell. Furthermore, the events so ordered had to satisfy certain constraints, notably that the output of a read operation equals the input to the write operation which most immediately preceded it (with respect to the particular total ordering).

The major difference between the entry-execution model and the actor model is in level of abstraction. The former is rooted in a view of programs as fixed sets of indivisible instructions. A single actor, on the other hand, can model anything from an addition operator to an entire program. An actor's function can change with time, and new actors can be created dynamically. Consequently, it is much harder to grasp the connection between a concrete data-flow program and an actor system which models it. Given the limited objective of this thesis, the entry-execution model seems more appropriate. It is felt, however, that partial orders of entries would be useful tools in specifying or proving what a program does.

5.2 The Heap Determined by a Computation

This section develops the definition of the heap determined by a computation (from an initial heap). The derivation confers the desirable property that for any firing sequence θ starting in any state $S = (\Gamma, U)$, the heap determined from U by the canonical computation $\eta(S, \theta)$ is the heap in the state $S \cdot \theta$. The significance of the concept is three-fold:

1. It demonstrates the relationship between an abstract computation and a more easily-visualized heap, without recourse to an interpreter.

2. It lies at the heart of the proof (in Section 7.3) that determinacy in the entry-execution model $EE(L_D, M)$ implies functionality of L_D programs.

3. It commences the verification that $EE(L_{BS}, S)$ is an S-S model.

The final five of the seven S-S constraints concern the values of the output entries of structure operation executions in a computation (or in a pair of computations) in a job J. The first and most difficult step in proving that these are satisfied by all computations in J is showing that they are satisfied by all canonical computations in J. The role in this of the heap determined by a computation is indicated in the following brief outline.

For any initial state $S = (\Gamma, U)$ and halted firing sequence Ω starting in S, for the canonical computation $\omega = \eta(S, \Omega)$:

1. The value of the output entries of a structure operation execution in ω equals the value of the tokens output by some firing φ in Ω.

2. The value of those tokens depends on the content of a particular node n in the heap in $S \cdot \theta$, where θ is such that $\theta\varphi$ is a prefix of Ω.

3. That heap is identical to the heap determined from U by $\eta(S,\theta)$.

4. The content of n in that latter heap depends on the initial content (in U) of some related node m, or on the inputs to certain executions in ω.

5. Those are just the executions whose durations in ω contain Ent(e,1).

Thus the values of the output entries of e may depend on the inputs to another execution e' in ω. It will be seen that there is such a dependence iff e is in the reach of e', and if so, the dependence will be that dictated by the Atomic or Structure Output Constraint. If e is in no reach, then its outputs depend just on the initial content of m. The outputs of another execution e' in another computation in the same job may depend in the same way on that initial content, if e' does not fall into a reach either. In this case, the outputs of the two executions will be equal, as required by the Initial Structure Constraint. Similar reasoning applies to the remaining two Constraints.

The first two subsections below describe the construction of a heap (N,Π,SM) from a computation: N and Π in Section 5.2.1 and SM in Section 5.2.2. Section 5.2.3 then proves that $\eta(S,\theta)$ determines the heap in $S \cdot \theta$.

5.2.1 Node Activation Records

A firing sequence θ starting in initial state $S = (\Gamma,U)$ determines a heap in the manner prescribed in Definition 2.3-1; this is the heap in the state being denoted as $S \cdot \theta$. The goal here is to define the heap determined from an initial heap by a computation in such a way that the heap determined from U by $\eta(S,\theta)$ is the heap in $S \cdot \theta$. This development is

complicated by the fact that there is not quite enough information in a computation to completely determine a heap from just U. The present subsection seeks to discover what information is missing from ω, and how it might best be supplied; the search commences by examining θ to see what information it uses to determine a heap.

A heap (Definition 2.2-2) is an ordered triple (N,Π,SM), where

N is a set of active nodes

Π is a one-to-one function from V_p onto N

SM is a function assigning a content to each node in N.

Let U be (N_0,Π_0,SM_0) and let the heap in $S \cdot \theta$ be (N,Π,SM). Then N consists of the nodes in N_0 plus those activated by Copy firings in θ. A Copy firing can activate any arbitrary node not already in the heap. Therefore, determining N requires explicitly specifying which nodes are activated by firings in θ. Similarly, Π consists of Π_0 plus an association of a unique pointer, also chosen arbitrarily, with each node in $N-N_0$. Hence, determining Π requires explicitly specifying, for each node activated by a Copy firing in θ, the pointer which points to it. Finally, the content $SM(n)$, for any $n \in N$, is determined from the initial content of a related node m, plus the inputs to certain firings in θ. As shown in Section 5.1.6, m is the unique node in N_0 from which n is dynamically descended in θ. Thus it is necessary to know, for each $n \in N-N_0$, which Copy firing activated n (to determine its corresponding m) and which pointer points to n (to determine Π).

This information was embedded in the firing sequence Ω by making

each firing φ of a Copy labelled C be $\varphi = (C,(p,n))$; this specifies

explicitly that the node activated by φ is n, and that the pointer to n

is p (i.e., $\Pi(p) = n$). The canonical computation $\omega = \eta(S,\Omega)$, however,

does not contain all of this information. For example, a computation

contains no nodes, so a separate listing of the nodes in $N-N_0$ will be

needed. There are pointers explicit in ω: Let p be the value of the

tokens output by the kth firing of Copy operator C in Ω. If one of

these is removed by a subsequent firing in Ω, then the execution $Ex(C,k)$

has output entries in ω with value p. However, if none of these tokens

is removed by firings in Ω (which is possible), then $Ex(C,k)$ has no

output entries in ω; i.e., even though p is in dom Π, it does not

necessarily appear in ω. Thus the only way to guarantee that the pointers

pointing to all of the nodes in $N-N_0$ are known is to supply a separate

list of them. Furthermore, each pointer in this list must be paired with

the node to which it points. The set of ordered pairs in $\Pi-\Pi_0$ contains

all of the above information.

Finally, in order to determine all dynamic descendancy relations,

the nodes in $N-N_0$ must be paired with the Copy executions initiated in ω.

All of the above lists and pairings can concisely be made explicit in

the form of a node activation record:

Definition 5.2-1 Given an interpretation Int, a domain V_p of pointers,

and a domain N of nodes, a <u>node</u> <u>activation</u> <u>record</u> is a function

$$NAR: C \to V_p \times N$$

where C is a set of executions of the Copy action, per Int.
Denote by "ran NAR" the multiset[†] containing, for each $C \in C$, the ordered
pair which is NAR(C).

$\triangle$

A node activation record NAR should contain enough additional
information for a computation ω to determine a heap from an initial heap.
Specifically, each Copy execution initiated in ω should be associated by
NAR with a pointer-node pair. These associations may in fact duplicate
information in ω: In addition to the pointer value associated with Copy
execution C by NAR, there may be pointer-valued output entries of C in ω;
if so, the two pointer values should of course be the same. Any node
activation record satisfying these two properties is compatible with ω:

<u>Definition 5.2-2</u> Given an interpretation Int and a computation ω for Int,
any node activation record NAR <u>is</u> <u>compatible</u> <u>with</u> ω iff, for every Copy
execution C initiated in ω with respect to Int:

1. NAR(C) is defined, and

2. if there are output entries of C in ω, then the value of those
   entries is the pointer in the ordered pair which is NAR(C).

$\triangle$

In the heap (N,Π,SM) determined by a computation ω from an initial
heap $U = (N_0,\Pi_0,SM_0)$ and a compatible node activation record NAR, Π is
formed by appending to Π_0 the multiset of pointer-node pairs in ran NAR.
Since any Π must be one-to-one, for each pair (p,n) in ran NAR, there can
be no (p',n) for $p' \neq p$ or (p,n') for $n' \neq n$ either in Π_0 or in ran NAR.

[†]multiset: Analogous to a set, but elements may appear more than once.
    [25, p. 627]

Furthermore, for each $(p,n) \in \Pi$, there must be a unique node m in N_0 such
that $SM(n)$ can be based on the initial content $SM_0(m)$. As explained in
Section 5.1.6, p is dynamically descended in ω from the pointer to m.
However, if the same pair (p,n) appears twice in ran NAR, or is both in
ran NAR and in Π_0, then p may be dynamically descended in ω from pointers
to two different nodes in N_0. This is because by compatibility, either
two different Copy executions may have output entries of value p, or p is
the value of the output entries of a Copy execution even though n itself
is in N_0. In order to simplify the determination of m, then, this ambigu-
ity, along with the possibility that Π will not be one-to-one, are
disallowed; i.e., the heap determined by ω from U and NAR is defined only
if ran NAR is consistent with U:

Definition 5.2-3 A multiset AP of pointer-node pairs is _consistent_ with
a heap $U = (N,\Pi,SM)$ iff no pointer or node in a pair in AP is also in any
other pair in either AP or Π.
$\triangle$

Given a compatible and consistent node activation record NAR, any
computation should be able to determine a new heap from any initial heap.
It is desired that, in particular, the computation $\eta(S,\theta)$, for any initial
state $S = (\Gamma,U)$ and firing sequence θ starting in S, should determine from
$U = (N_0,\Pi_0,SM_0)$ the heap (N,Π,SM) in $S'\theta$. This requires that ran NAR
should equal the set of pointer-node pairs $\Pi-\Pi_0$, which is in turn just the
set of ordered pairs in the Copy firings in θ. If the kth firing of Copy
actor d in θ is $(d,(p,n))$, then that firing places tokens of value p
d's output arcs (Definition 2.3-1). Then if Copy execution $Ex(d,k)$ has

output entries in $\eta(S,\theta)$, they have value p (Lemma 4.3-1). For compatibility, therefore, the appropriate node activation record is one which associates (p,n) with $Ex(d,k)$:

__Definition 5.2-4__ Given any L_{BS} program P, let θ be any firing sequence starting in an initial state for P, and let ω be any computation for Int(P). Then the node activation record derived from θ and ω, NAR, is specified as follows:

1. NAR(C) is defined iff C is a Copy execution initiated (wrt Int(P)) in ω.

2. For any such Copy execution C, let d and k be the label and integer such that $C = Ex(d,k)$. Then NAR(C) = (p,n), where the kth firing of d in θ is $(d,(p,n))$.

$\triangle$

Lemma 5.2-2 below verifies that the node activation record derived from θ and $\eta(S,\theta)$ is compatible with $\eta(S,\theta)$, and that its range is consistent with the heap in S. This is preceded by a confirmation of the cumulative effect of Copy firings in a firing sequence.

__Lemma 5.2-1__ Let S be any initial (standard or modified) state for an L_{BS} program, and let Ω be any firing sequence starting in S. Let $\theta\varphi$ be any prefix of Ω in which φ is a Copy firing $(d,(p,n))$. Then for any prefix Δ of Ω,

$$|\Delta| < |\theta\varphi| \implies p \notin \text{dom } \Pi \text{ and } n \notin N \text{ in } S \cdot \Delta, \text{ and}$$

$$|\Delta| \geq |\theta\varphi| \implies p \in \text{dom } \Pi \text{ and } n \in N \text{ in } S \cdot \Delta.$$

__Proof:__ Prove the second implication first, by induction on the length of Δ.

Basis: $|\Delta| = |\theta\varphi|$.

(1) (p,n) is added to Π in going from $S\cdot\theta$ to $S\cdot\theta\varphi = S\cdot\Delta$ Def. 2.3-1

(2) p∈dom Π and n∈N in $S\cdot\Delta$ (1)+Def. 2.2-5

Induction step: Assume the second part of the Lemma is true for the length-k prefix Δ of Ω, $|\theta\varphi| \leq k < |\Omega|$ and consider the prefix $\Delta\varphi'$ of length k+1.

(3) p∈dom Π and n∈N in $S\cdot\Delta$ ind. hyp.

(4) dom Π in $S\cdot\Delta$ is a subset of dom Π in $S\cdot\Delta\varphi'$ and N in $S\cdot\Delta$ is a subset

     of N in $S\cdot\Delta\varphi'$ Def. 2.2-5

(5) p∈dom Π and n∈N in $S\cdot\Delta\varphi'$ (3)+(4)

Thus it is proven by induction that

(6) $|\Delta| \geq |\theta\varphi|$ $\Rightarrow$ p∈dom Π and n∈N in $S\cdot\Delta$

Now prove the first part of the Lemma by contradiction. Assume

(7) There is a prefix Δ of Ω, $|\Delta| < |\theta\varphi|$, such that p∈dom Π or n∈N in $S\cdot\Delta$

By the induction above, since $|\theta| \geq |\Delta|$,

(8) p∈dom Π or n∈N in $S\cdot\theta$ (2)+(6)

(9) (p,n) cannot be added to Π in going from $S\cdot\theta$ to $S\cdot\theta\varphi$ (8)+Def. 2.2-5

Since (7) leads to a contradiction between (1) and (9), (7) is false. $\triangle$

Lemma 5.2-2 Given an L_{BS} program P, let $S = (\Gamma,U)$ be any initial state for P. Let Ω be any firing sequence starting in S, and let $\omega = \eta(S,\Omega)$. Then the node activation record derived from Ω and ω is meaningfully defined and is compatible with ω, and ran NAR is consistent with U.

Proof:

(1) ω is a computation for Int(P)   lemma 4.3-3

(2) Let $\text{Int}(P) = (\text{St},I,\text{IE})$. The definition of NAR makes sense iff, for

    every execution $C = \text{Ex}(d,k)$ initiated in ω wrt $\text{Int}(P)$, $I(d) = \text{Copy}$

    $\Rightarrow$ there are at least k firings of d in Ω (1)+Def. 5.2-4

(3) C is initiated in ω and $I(d) = \text{Copy} \Rightarrow$ there is $\text{In}(\text{Copy}) = 1$ input

    entry to C in ω Defs. 4.3-1+4.2-6

(4) NAR is meaningfully defined   (3)+(2)+Lemma 4.3-1

(5) For every Copy execution C initiated in ω, $\text{NAR}(C)$ is defined

                                            Def. 5.2-4

(6) Let (p_1,n_1) and (p_2,n_2) be any two distinct pairs in the multiset

    ran NAR. Let C_1 and C_2 be the two Copy executions such that, for

    $i=1,2$, $\text{NAR}(C_i) = (p_i,n_i)$, and let d_i and k_i be such that $C_i =$

    $\text{Ex}(d_i,k_i)$. Then (p_i,n_i) is in the k_i^{th} firing in Ω which contains

    d_i Def. 5.2-4

(7) The k_1^{th} firing of d_1 and the k_2^{th} firing of d_2 are distinct firings

    in Ω. Assume, without loss of generality, that the k_2^{th} firing of

    d_2 is the later of these. I.e., there is a prefix $\theta\varphi_1\Delta\varphi_2$ of Ω

    in which $\varphi_i = (d_i,(p_i,n_i))$ (6)

(8) Let the heap in $S \cdot \theta\varphi_1$ be (N,Π,SM) and let U be (N_0,Π_0,SM_0). Then

    since $|\theta\varphi_1| < |\theta\varphi_1\Delta\varphi_2|$ and $|\lambda| < |\theta\varphi_1\Delta\varphi_2|$, $p_2 \notin \text{dom } \Pi$, $n_2 \notin N$,

    $p_2 \notin \text{dom } \Pi_0$, and $n_2 \notin N_0$ (7)+Lemma 5.2-1

(9) $p_1 \in \text{dom } \Pi$ and $n_1 \in N$, so $p_2 \neq p_1$ and $n_2 \neq n_1$ (8)+(7)+Defs. 2.3-1+2.2-5

(10) For any pair in ran NAR, neither the pointer nor the node in that

    pair is in any other pair in either ran NAR or Π_0; i.e., ran NAR

    is consistent with U (6)+(8)+(9)+Def. 5.2-3

(11) Let C be any Copy execution initiated in ω, and let d and k be such

that $C = Ex(d,k)$. C has output entries in $\omega \Rightarrow$ there is a prefix $\Delta\varphi$ of Ω containing exactly k firings of C such that tokens of value p appear on the output arcs of the actor labelled d in P at the transition from $S \cdot \Delta$ to $S \cdot \Delta\varphi$ Lemma 4.3-1

(12) $\twoheadrightarrow$ that transition is the result of the kth firing of d in Ω, and $\exists n \in N$: (p,n) is added to Π at that transition   Def. 2.2-5

(13) $\Rightarrow$ (p,n) is in that firing   Def. 2.3-1

(14) $\Rightarrow$ NAR(C) = (p,n)   (11)+Def. 5.2-4

(15) If C has output entries in ω, their value is the pointer in NAR(C)

     (11)+(14)

(16) NAR is compatible with ω (1)+(5)+(15)+Def. 5.2-2

$\triangle$

The following definition of a *quasi-inverse of a node activation record* will prove very convenient:

<u>Definition 5.2-5</u> Given a node activation record

$$\text{NAR}: C \to V_p \times N$$

the <u>Creating-Copy</u> <u>function corresponding to</u> NAR,

$$\text{CC}: V_p \to C$$

is given by

$$\text{CC(p)} = \begin{cases} C & \text{if there is an } n \in N \text{ such that NAR(C)} = (p,n) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$\triangle$

For any pointer p, CC(p) is the Copy execution whose output entries have value p, as the following lemma shows:

<u>Lemma 5.2-3</u> Let S be any initial standard state for an L_{BS} program P, and let Ω be any firing sequence starting in S. Let η be the canonical computation $\eta(S,\Omega)$, and let the heap in S be (N,Π,SM). Let NAR be the node activation record derived from Ω and η, and let CC be the corresponding Creating-Copy function. For any pointer p which is the value of an entry in η:

p is the value in η of the output entries of a Copy execution

$\Rightarrow$ p $\notin$ dom Π

$\Rightarrow$ CC(p) is defined, the first entry in η with value p is an output entry of CC(p), that entry is strictly preceded by Ent(CC(p),1), and there is no other Copy execution whose output entries have value p.

<u>Proof</u>: Prove the second implication first.

(1) Let p be any pointer not in dom Π which is the value of some entry in η. Then either some firing in Ω removes a token with value p or Ω is halted and there is a token with value p on an arc in
$S \cdot \Omega$ Alg. 4.3-1

In what follows, alternatives in parentheses refer to the case that Ω is halted and no firing in it removes a token of value p.

(2) Let $\theta\varphi$ be the prefix of Ω such that φ is the first firing (if there is one) in Ω to remove a token with value p. Let b be an arc of P from which such a token is removed by φ (or let b be an arc on which there is a token of value p in $S \cdot \Omega$). Either that token is on b in S or it is placed there by some firing φ' in θ (or in Ω) of the actor d of which b is an output arc   (1)+Def. 2.1-5

(3) If that token in on b in S, then p is in dom Π Def. 2.2-6

(4) That token is placed on b by a firing φ' of d in θ (or Ω)(1)+(3)+(2)

(5) Let Δ be such that $\Delta\varphi'$ is a prefix of θ (or Ω). φ' does not

     remove a token with value p   (4)+(2)

(6) d is either a Select or a Copy operator   (4)+(5)+Defs. 2.2-5+2.2-4

(7) Let n be $\Pi(p)$. Since $p \notin$ dom Π, $n \notin N$, so there is no node m and

     selector s such that $(s,n) \in SM(m)$ in S Def. 2.2-1

(8) For any prefix $\Xi\varphi''$ of Δ, if there is a node m and selector s such

     that $(s,n) \in SM(m)$ in $S \cdot \Xi\varphi''$, then either (s,n) is in $SM(n)$ in $S \cdot \Xi$ or

     φ'' is an Update firing which removes a token of value p Table 2.2-1

Thus by induction, since no firing in Δ removes a token of value p,

(9) There is no m and s such that $(s,n) \in SM(m)$ in $S \cdot \Delta$ (2)+(5)

(10) d is a Select operator $\Rightarrow$ there is a node m and selector s such

      that $(s,n) \in SM(m)$ in $S \cdot \Delta$ Table 2.2-1

(11) d is not a Select operator   (9)+(10)

(12) d is a Copy operator   (6)+(11)

(13) There is no entry with value p in $\omega(S,\theta)$ (or $\omega(S,\Omega)$)   (2)+Alg. 4.3-1

(14) Letting $Int(P) = (St, I, IE)$, there is an entry with value p in

      $\omega(S,\theta\varphi)$ (or $\eta(S,\Omega)$), that is an output entry of execution $Ex(d,k)$

      for some k, and $I(d)$ = Copy   (2)+(4)+(12)+Alg. 4.3-1+Def. 4.3-2

(15) $\omega(S,\theta\varphi)$ (or $\eta(S,\Omega)$) is a prefix of η Alg. 4.3-1

(16) The first entry in η with value p is a output entry of a Copy

      execution $C = Ex(d,k)$ (13)+(14)+(15)

(17) η is causal with respect to $Int(P)$ Lemma 4.3-2

(18) The initiating entry to C strictly precedes the first entry in η

with value p, so C is initiated in η (17)+(16)+Def. 4.2-7

(19) Since d has only a number-1 input arc, that initiating entry can

only be Ent(C,1)   Alg. 4.3-1

(20) NAR is compatible with η and ran NAR is consistent with the heap

in S Lemma 5.2-2

(21) There is a node n such that (p,n) = NAR(C)   (20)+(18)+(16)+Def. 5.2-2

(22) CC(p) is defined and equals C   (21)+Def. 5.2-5

(23) There is no other Copy execution C' and node n' such that

(p,n') = NAR(C')   (20)+Defs. 5.2-3+5.2-4

(24) There is no other Copy execution of which p is the value of the

output entries   (20)+(23)+Def. 5.2-2

Now prove that if p is the value in η of the output entries of a Copy

execution C = Ex(d,k), then p$\notin$dom Π.

(25) There is a node n such that (p,n) = NAR(C)   (16)-(21)

(26) The kth firing of d in Ω is (d,(p,n))   (25)+Def. 5.2-4

(27) Let $\Delta\varphi$ be any prefix of Ω such that φ = (d,(p,n)). Then since

$|\lambda| < |\Delta\varphi|$, p$\notin$dom Π in $S\cdot\lambda$ = S (26)+Lemma 5.2-1+Def. 2.3-1

$$\triangle$$

5.2.2 The Contents Determined by a Computation

With the additional information supplied by a compatible and consist-
ent node activation record NAR, a computation ω can determine a heap
(N,Π,SM) from an initial heap $U = (N_0,\Pi_0,SM_0)$. N and Π are constructed
directly from NAR as follows: Π is Π_0 plus the pairs in ran NAR, and N is
the set of nodes in the ordered pairs in Π. All that remains is to
establish how, for each node n$\in$N, ω determines the content SM(n) from U

and NAR. There are two cases to consider: a given content either is or is not dependent on some execution(s) in ω.

5.2.2.1 Contents Dependent on Executions in ω

Let Ω be any firing sequence starting in any initial state $S = (\Gamma, U)$, and let $\Delta\varphi$ be any prefix of Ω. The intent here is that the heap determined from U by $\alpha = \eta(S, \Delta)$ should be the heap (N,Π,SM) in $S^{\cdot}\Delta$. Assume that the last firing φ in $\Delta\varphi$ is the kth firing of Fetch operator d, and that φ accesses node n. Two conclusions can be drawn about the value v output by φ: (1) it is the value in SM(n) (Table 2.2-1) and (2) it is the value of the output entries in $\omega = \eta(S, \Omega)$ of $F = Ex(d,k)$ (Lemma 4.3-1). Therefore, α should determine that the value in SM(n) is the value of the output entries of F in ω.

It has already been argued (Section 5.1.4) that if $f = Ent_{\omega}(F,1)$ falls into the duration D(A) of some Assign execution A, then the output entries of F have value $V(Ent_{\omega}(A,2))$. If f is in D(A), then, letting $p = V(f)$, either:

   (a) $Ent_{\omega}(A,1)$ is the last input entry of an Assign execution preceding
      f in H_p^{ω}, or

   (b) there is no input entry to an Assign execution preceding f in H_p^{ω},
      there is a Copy execution C which has output entries of value p in
      ω, and $Ent_{\omega}(C,1)$ is in D(A) (or alternatively, C is in the reach
      R(A) in ω).

A simple inductive argument shows that A initiates before F: In case (a) above, A must initiate before F by definition of access history, and in case (b), C must initiate before F (Lemma 5.2-3) and A must initiate

before C by induction hypothesis. Therefore, A can be identified from the shortest prefix of ω in which F is initiated; this is $\beta = \eta(S, \Delta\varphi)$.

The goal is to determine SM(n) (which requires finding A) from just $\alpha = \eta(S, \Delta)$. Entry f is not in any access history in α, and so A cannot be identified as the Assign execution whose duration contains f. Since β is α followed by the input entries to F, however, F is the last execution initiated in β. Therefore, H_p^β is H_p^α followed by f; i.e., H_p^α is the prefix of H_p^ω preceding f. From this and the above, f is in D(A) in ω iff:

(a) $\text{Ent}_\alpha(A,1)$ is the last input entry to an Assign execution in H_p^α, or

(b) there is no input entry to an Assign execution in H_p^α, there is a Copy execution C which has output entries of value p in ω, and C is in R(A) in α.

This characterization of A still relies on information which may not be in α: It is possible that f is the first entry in ω with value p. If so, then there is no Assign input entry in H_p^α, and the Copy execution C has no output entries in α. Given a node activation record NAR which is compatible with ω, however, it is known that if any Copy execution has output entries in ω of value p, it is the one which created p, CC(p). Thus it is possible to identify, from just α and NAR, any Assign execution A whose duration in ω contains $\text{Ent}_\omega(F,1)$. It will be said that this duration of A "extends to the end of H_p^α" (even though H_p^α may be empty):

<u>Definition 5.2-6</u> (Durations extending to the end of an access history) Given any interpretation Int, computation α for Int, and node activation record compatible with α, let CC be the Creating-Copy function corresponding to that node activation record. Denote by AS the set of Assign

executions initiated in α, and by SS the set of all Update and Delete

executions initiated in α. Denote by APS the set $\{\text{Ent}_\alpha(e,1) \mid e \in AS\}$, and

for each $U \in SS$, denote by SPS(U) the set

$\{\text{Ent}_\alpha(e,1) \mid e \in SS \text{ and } V(\text{Ent}_\alpha(e,2)) = V(\text{Ent}_\alpha(U,2))\}$.

For each $A \in AS$ and pointer p, the <u>duration $D(A)$ extends to the end of</u>

$\underline{H_p^\alpha}$ iff:

1. $\text{Ent}_\alpha(A,1)$ is the last entry from APS in H_p^α, or

2. There is no entry from APS in H_p^α, and CC(p) is defined and is in

   R(A) in α.

For each $U \in SS$ and pointer p, the <u>duration $D(U)$ extends to the end of</u>

$\underline{H_p^\alpha}$ iff:

1. $\text{Ent}_\alpha(U,1)$ is the last entry from SPS(U) in H_p^α, or

2. There is no entry from SPS(U) in H_p^α, and CC(p) is defined and is in

   R(U) in α.

$\triangle$

The foregoing argument can be summarized thusly: Given a firing

sequence Δ starting in $S = (\Gamma, U)$, it is desired that the computation

$\alpha = \eta(S, \Delta)$ should determine from U the heap (N, Π, SM) in $S \cdot \Delta$. For any

pointer p, let n be $\Pi(p)$, and assume that there is an Assign execution A

such that $D(A)$ extends to the end of H_p^α. The following chain of infer-

ences can then be drawn:

There is a firing sequence $\Delta\varphi$ in which φ is the kth firing of Fetch

operator d and φ's pointer input is p

$\rightarrow \varphi$ outputs the value in SM(n), and there is a computation $\beta = \eta(S, \Delta\varphi)$

in which, for Fetch execution $F = \text{Ex}(d,k)$, $\text{Ent}_\beta(F,1)$ falls into $D(A)$

(Lemma 5.2-7 below), so that F is in R(A) in β

⟹ F is in R(A) in any computation ω of which β is a prefix

⟹ if F has output entries in ω, their value equals the value output

   by φ, and their value equals $V(Ent_\omega(A,2))$

⟹ the value in SM(n) is $V(Ent_\omega(A,2)) = V(Ent_\alpha(A,2))$.

SM(n) cannot depend on what firings may or may not occur after Δ.

Therefore, the conclusion is that if D(A) extends to the end of H_p^α,

then the value in $SM(\Pi(p))$ is $V(Ent_\alpha(A,2))$.

5.2.2.2 Contents Not Dependent on Executions in ω

It remains now to consider the case of a pointer p for which there

is no Assign execution whose duration extends to the end of H_p^α. In this

case, a similar inference can be drawn, with the aid of the argument

advanced in Section 5.1.6:

There is a firing sequence $\Delta\varphi$ in which φ is the kth firing of

Fetch operator d and φ's pointer input is p

⟹ in any computation ω of which $\eta(S,\Delta\varphi)$ is a prefix and in which

Fetch execution F = Ex(d,k) has output entries, their value equals

the value in SM(n) and their value equals the value in $SM_0(m)$, where

$(q,m) \in \Pi_0$ and $DD_\omega(q,p)$; i.e., p is dynamically descended from q in ω.

This determination of the value in SM(n) suffers from a familiar

shortcoming: The goal is to determine SM(n) from just $\alpha = \eta(S,\Delta)$, U,

and a node activation record. It is required to discover the particular

$q \in dom\ \Pi_0$ from which p is dynamically descended in ω (Lemma 5.2-4 below

proves that q is unique in dom Π_0). From Definition 5.1-9, DD(p,p) in

any computation; thus, if p is in dom Π_0, then q = p. Otherwise, if there

are entries in α with value p, then $DD_\omega(q,p)$ iff $DD_\alpha(q,p)$. But it is

possible that there are no such entries in α; in this case, it is meaning-less to speak of p's being dynamically descended in α from any pointer in dom Π_0. Therefore, q cannot be defined as the unique pointer in dom Π_0 such that $DD_\alpha(q,p)$.

Fortunately, q can be determined, in an indirect manner, from just α, U, and the node activation record NAR derived from Δ and α: Let β be $\eta(S,\Delta\varphi)$, let NAR' be the node activation record derived from $\Delta\varphi$ and β, and let CC_α and CC_β be the Creating-Copy functions corresponding to NAR and NAR'. The computation β consists of α followed by $Ent_\beta(F,1)$, which entry has value p. If $p\notin dom\ \Pi_0$, then p is the value in β of the output entries of Copy execution $C = CC_\beta(p)$, and C initiates in α (Lemma 5.2-3). Since NAR' is compatible with β (Lemma 5.2-2), NAR'(C) is defined and equal to the pair (p,n), for some n. NAR(C) is also defined and equal to NAR'(C) (Lemma 5.2-5 below). Therefore, $CC_\alpha(p) = CC_\beta(p) = C$.

Thus if $p\notin dom\ \Pi_0$, it is the value in β (hence in ω) of the output entries of the Copy execution $CC_\alpha(p)$, which execution can be identified from just NAR. Let p' be the value of $Ent_\alpha(CC_\alpha(p),1)$; for any $q\in dom\ \Pi_0$, $DD_\omega(q,p)$ iff $DD_\omega(q,p')$ iff $DD_\alpha(q,p')$. Therefore, if $p\notin dom\ \Pi_0$, q is determined to be the unique pointer in dom Π_0 from which $V(Ent_\alpha(CC_\alpha(p),1))$ is dynamically descended in α.

The derivation just given is summarized in the following sub-section as the definition of the heap determined by a computation from an initial heap and a compatible and consistent node activation record. It is then proven that, for initial state $S = (\Gamma,U)$ and firing sequence Δ starting in S, $\alpha = \eta(S,\Delta)$ determines the heap in $S\cdot\Delta$ from U and the node activation record derived from Δ and α.

5.2.3 Summary and Validation

The following definition assumes that for each pointer, there is a unique pointer in dom Π_0 from which it is dynamically descended; the validity of this assumption is confirmed immediately after the definition.

<u>Definition 5.2-7</u>   Given any heap $U = (N_0, \Pi_0, SM_0)$ and any interpretation Int, let α be any computation for Int and let NAR be any node activation record such that NAR is compatible with α and ran NAR is consistent with U.   Then the <u>heap determined by α from U and NAR</u>, $(N_\alpha, \Pi_\alpha, SM_\alpha)$, is defined as follows:

$$\Pi_\alpha = \Pi_0 \text{Uran NAR}$$
$$N_\alpha = \{n \mid \exists p: (p,n) \in \Pi_\alpha\}$$

Let CC be the Creating-Copy function corresponding to NAR. For each pair $(p,n) \in \Pi_\alpha$, let (q,m) be defined as follows:

a.   If $(p,n) \in \Pi_0$, then $(q,m) = (p,n)$.

b.   If $(p,n) \notin \Pi_0$, then (q,m) is that unique pair in Π_0 such that $V(Ent_\alpha(CC(p),1))$ is dynamically descended from q in α.

Then $SM_\alpha(n)$ is given by:

1.   If there is an Assign execution A such that D(A) extends to the end of H_p^α, then the value in $SM_\alpha(n)$ is $V(Ent_\alpha(A,2))$. Otherwise, it is the value in $SM_0(m)$.

2.   For each selector $s \in \Sigma$, if there is an Update execution U such that D(U) extends to the end of H_p^α, then the pair $(s, \Pi_\alpha(V(Ent_\alpha(U,3))))$ is in $SM_\alpha(n)$, and is the only pair in $SM_\alpha(n)$ containing s. If there is a Delete execution U such that D(U) extends to the end of H_p^α, then

there is no pair containing s in $SM_\alpha(n)$. Otherwise, for any node r,
$(s,r) \in SM_\alpha(n)$ iff $(s,r) \in SM_0(m)$.

$\triangle$

Lemma 5.2-4 Let $S = (\Gamma, U)$ be any initial state for an L_{BS} program
P, where $U = (N, \Pi, SM)$. Let Δ be any firing sequence starting in S, and
let $\alpha = \eta(S, \Delta)$. Then for any pointer p which is the value of some entry
in α, there is a unique $q \in \text{dom } \Pi$ such that $DD_\alpha(q,p)$.

Proof: By contradiction. Assume that

(1) The lemma is false

(2) There is a prefix γf of α such that any pointer which is the value
   of an entry in γ is dynamically descended from a unique pointer
   in dom Π, but $p = V(f)$ is dynamically descended from two distinct
   pointers in dom Π (1)

(3) p is the value of the output entries of a C oy execution $\Rightarrow$
   $p \notin \text{dom } \Pi$ Lemma 5.2-3

(4) $p \in \text{dom } \Pi \Rightarrow$ p is not the value of the output entries of a Copy
   execution   (3)

(5) $\Rightarrow$ p is dynamically descended only from itself   Def. 5.1-9

(6) $\Rightarrow$ p is dynamically descended from a unique pointer in dom Π (4)

(7) f is the first entry in α with value p   (2)

(8) $p \notin \text{dom } \Pi \Rightarrow$ f is an output entry of a Copy execution C, $g = \text{Ent}_\alpha(C,1)$
   strictly precedes f in α (i.e., g is in γ), and no other Copy
   execution has output entries in α of value p   (7)+Lemma 5.2-3

(9) Let $r = V(g)$. Then for any $q \neq p$, $DD_\alpha(q,p) \Rightarrow DD_\alpha(q,r)$ (8)+Def. 5.1-9

Since r is the value of an entry in γ

(10) (2) $\Rightarrow$ r is dynamically descended from a unique pointer in dom Π

$$(9)+(8)$$

Since (2) $\Rightarrow$ p is dynamically descended from two pointers in dom Π,

(11) (2) $\Rightarrow$ r is dynamically descended from two pointers in dom Π (8)+(9)

Since (2) implies a contradiction between (10) and (11), (1) is false, and

the Lemma is true.

$\triangle$

The proof that $\eta(S,\Delta)$ determines the heap in $S \cdot \Delta$ is by induction on

the lengths of the prefixes of Δ. Therefore, it is necessary first to

establish that, for any firing sequence $\theta\varphi$, the key entities — reaches,

access histories, and node activation records — derived from θ and $\eta(S,\theta)$

are subsets of (i.e., agree with) those derived from $\theta\varphi$ and $\eta(S,\theta\varphi)$. This

is easily done for the latter two entities in the following.

__Lemma 5.2-5__ Let S be any initial standard state for an L_{BS} program P, and

let $\theta\varphi$ be any firing sequence starting in S. Let the last firing in $\theta\varphi$,

φ, be the kth firing of an actor in P labelled d. Let $\alpha = \eta(S,\theta)$ and

$\beta = \eta(S,\theta\varphi)$. Then

   A: For any pointer p, H_p^α is a prefix of H_p^β, any input entries to Ex(d,k)

        which have value p are in H_p^β, and for any entry Ent(e',j) in $H_p^\beta - H_p^\alpha$,

        e' $\neq$ Ex(d,k) $\Rightarrow$ e' is not a structure operation execution.

   B: Let NAR (NAR') be the node activation record derived from θ and α

        ($\theta\varphi$ and β). Then for any Copy execution C initiated in α,

        NAR'(C) = NAR(C).

Proof:

(1) β is α followed by m input entries to $Ex(d,k)$, where φ removes m

    tokens, followed possibly by input entries to executions $Ex(c,n)$

    where $c \in DL$ Alg. 4.3-1+Def. 4.3-1

(2) The set of entries in α whose values are p is a subset of the set

    of entries in β whose values are p, and for any j less than or

    equal to the number of executions initiated in α, the jth

    execution initiated in α is the jth execution initiated in β

                                                                 (1)+Def. 4.2-6

(3) H_p^{α} is a prefix of H_p^{β} (2)+Def. 5.1-4

(4) Letting $Int(P) = (St, I, IE)$, $m = In(I(d))$ (1)+Defs. 4.3-2+4.3-1

(5) $Ex(d,k)$ is initiated in β, so any input entries to it which have

    value p are in H_p^{β} (1)+(4)+Defs. 4.2-6+5.1-4

(6) For any entry $f = Ent(e',j)$ where $e' = Ex(c,n)$, $f \in H_p^{\beta} - H_p^{\alpha} \Rightarrow e'$ is

    initiated in β but not in α (2)+Def. 5.1-4

(7) $\Rightarrow [e' \neq Ex(d,k) \Rightarrow I(c)$ is not a structure operation]   (1)+Def. 4.3-2

(8) Let NAR (NAR') be the node activation record derived from θ and α

    ($\theta\varphi$ and β). NAR is meaningfully defined   Lemma 5.2-2

(9) Let $C = Ex(d,k)$ be any Copy execution initiated in α. Then there

    are k firings of d in θ and the kth of these is $(d,(p,n))$, where

    $(p,n) = NAR(C)$ (8)+Def. 5.2-4

(10) There are k firings of d in $\theta\varphi$, and the kth of these is $(d,(p,n))$,

    so $NAR'(C) = (p,n) = NAR(C)$ (9)+Def. 5.2-4

$\triangle$

    The need to show that the reach of an execution in one computation α

is a subset of its reach in another computation β arises in several

situations in the remainder of the thesis. In two of these, α is strongly related to β: either it is a prefix or it is a permutation which preserves initiating order of executions. The weakest relation obtains in the case that $\alpha = \eta(S,\Omega)$ and $\beta = \eta(S',\Omega)$, where S is any initial modified state, S' is the corresponding initial standard state, and Ω is any halted firing sequence starting in S. In the interest of efficiency, these needs are anticipated here by a single, general proof applicable in all three cases.

A study of the definitions of reach and duration reveals the following sufficient conditions for the reaches in α to be subsets of those in β:

1. Every structure operation execution which has input entries in α has input entries of the same values in β.

2. For any two structure operation executions e and e' such that $Ent_\alpha(e,1)$ is in an access history in α, $Ent_\alpha(e',1)$ precedes $Ent_\alpha(e,1)$ in that history iff $Ent_\beta(e',1)$ precedes $Ent_\beta(e,1)$ in an access history in β.

3. Every Copy execution which has output entries in either α or β has output entries of the same value in both computations.

Given the first of these conditions, the second is in turn guaranteed if:

2'. For every structure operation execution e initiated in α,

   a. e is initiated in β, and

   b. for any other structure operation execution e', e' initiates before e in α iff it does so in β.

If these sufficient conditions hold for α and β, then β is structure-operation-execution inclusive of α, as defined formally next.

Definition 5.2-8 For any two computations α and β for the same interpretation Int $=$ (St, I, IE), β is <u>structure-operation-execution</u> <u>inclusive</u> (<u>SOE-inclusive</u>) of α iff the following are all true (all initiations are with respect to Int):

1. Any structure operation execution initiated in α is initiated in β.

2. For any two structure operation executions e and e' such that e is initiated in α, e' is initiated before e in α iff e' is initiated before e in β.

3. For any Copy execution C initiated in α, C has output entries in β only if C has output entries in α.

4. For every entry $f \in \alpha$, there is an entry with the same value in β whose transfer has the same source as $T(f)$.

5. For any non-pI execution e and for any j, if there is an entry $Ent_\alpha(e,j)$ in α, then there is an entry $Ent_\beta(e,j)$ in β with the same value.

$\triangle$

The following lemma states the general result that the reaches in α are subsets of those in β if either α is a prefix of β or β is SOE-inclusive of α. Since β is SOE-inclusive of any of its permutations which preserve initiation order (Lemma 5.3-7), this covers all of the cases cited above. To enable a simple proof by induction on the lengths of the prefixes of α, a further requirement is imposed: For any pointer p which is the value in β of the output entries of a Copy execution C, the initiation of C precedes in β the initiation of any e such that $Ent_\beta(e,1)$

is in H_p^β. If β is causal, this can be simplified to "the first entry in β with value p is an output entry of C", since that entry must be preceded by C's initiating entry. (Any canonical computation β meets this requirement by Lemma 5.2-3.)

Lemma 5.2-6 Let α and β be any two causal computations for the same interpretation Int such that either α is a prefix of β or β is SOE-inclusive of α, and for any pointer p, p is the value of the output entries in β of a Copy execution C only if the first entry in β with value p is an output entry of C. Let e be any structure operation execution initiated in α wrt Int. Then for any Assign, Update, or Delete execution A, e is in R(A) in β iff e is in R(A) in α only if A is initiated in α.

Proof: (As with the succeeding Lemma and Theorem 5.2-1, the proof of this is essentially a tedious manipulation of definitions; therefore, all three proofs may be found in Appendix D.)

$\triangle$

The next lemma verifies the key property claimed for a duration extending to the end of an access history:

Lemma 5.2-7 Let S be any initial standard state, and let $\theta\varphi$ be any firing sequence starting in S in which the last firing is φ. Let $\alpha = \eta(S,\theta)$ and $\beta = \eta(S,\theta\varphi)$. Let f be any entry in β but not in α whose value is some pointer p. If $f = Ent_\beta(e,1)$ for some execution e, then for any other execution e', f is in duration D(e') in β iff D(e') extends to the end of H_p^α. Furthermore, if $\theta = \lambda$, then no durations extend to the end of H_p^α for any p.

$\triangle$

Now it is straightforward (if tedious) to prove formally that the foregoing construction was correct:

Theorem 5.2-1 Let $S = (\Gamma, U)$ be any initial standard state for an L_{BS} program P, and let Ω be any firing sequence starting in S. Let ω be $\eta(S, \Omega)$ and let NAR be the node activation record derived from Ω and ω. Then the heap determined by ω from U and NAR is defined and is identical to the heap in the state $S \cdot \Omega$.

$\triangle$

5.3 Validation of the S-S Model

The constraints defining an S-S model were constructed in such a way that $EE(L_{BS}, S)$ would satisfy them, which would in turn mean that those constraints do define, in the sense being used here, the set of structure operations in L_{BS}. The purpose of this section is to confirm the validity of the construction by means of a rigorous proof, the principle of which is briefly explained next.

In the five-tuple (V, L, A, In, E) which is $EE(L_{BS}, S)$, V, A, and In obviously meet the requirements imposed on them by the definition of an S-S model (Definition 5.1-1). All that remains is to show that every job from every expansion in E satisfies the seven constraints. The first of these, the Input/Output Type Constraint, is trivial; the second one, Pointer Transparency, is a straightforward special case which will not be discussed here (the proofs that these two constraints are satisfied are in Section 5.3.1 below).

The remaining five constraints all fit one of three patterns:

(1) The values of the output entries of an execution e_1 in any

computation $\dot{\omega}$ must depend on the value of an input entry in ω to another, related execution e_2.

(2) The output entries of one execution e_1 must be unequal to the output entries of another, related execution e_2 in the same computation ω.

(3) The values of the output entries of two related executions e_1 and e_2 in two computations ω_1 and ω_2 in the same job must be equal.

In all of these cases, the qualifying relationship between e_1 and e_2 is based partially on the actions of which they are executions. The remainder of the relation in patterns (1) and (2) may involve the concept of reach: a constraint of type (1) applies to e_1 and e_2 only if e_1 is in the reach $R(e_2)$ in ω, while in pattern (2), e_2 may have to fall in no reach. Those constraints fitting pattern (3) combine reach with the equal-pointer relation ρ: the constraints apply only if (a) either e_1 and e_2 are both *in* no reach or they both are in identical sets of reaches, and (b) letting p_1 and p_2 be the pointer inputs to e_1 and e_2, $(p_1,\omega_1)\rho(p_2,\omega_2)$.

Every computation α in a job J is a prefix of a halted computation which is a permutation of a canonical computation $\omega \in J$. The approach taken here is to prove first that the constraints are satisfied by every canonical computation, or pair of canonical computations, as appropriate, in J; this step makes extensive use of the just-defined heap determined by a computation. Then it is shown that, for any two computations α_1 and α_2 in J, there are two canonical computations ω_1 and ω_2 in J such that the pertinent qualifying relations between executions in α_1 and α_2 are subsets of (i.e., agree with) those in ω_1 and ω_2, respectively. That is:

A: For $i=1,2$, for any execution e_1 initiated in α_i, e_1 is in reach $R(e_2)$ in α_i iff e_1 is in $R(e_2)$ in ω_i.

B: For any two pointers p_1 and p_2, $(p_1,\alpha_1)\rho(p_2,\alpha_2) \Rightarrow (p_1,\omega_1)\rho(p_2,\omega_2)$.

This second step is accomplished with the aid of an intermediate computation: For $i=1,2$, α_i is a prefix of a halted computation β_i in J; there is in turn a canonical computation ω_i in J which is a permutation of β_i preserving initiation order (and hence is SOE-inclusive of β_i). Implication A holds for α_i and β_i, and then again for β_i and ω_i, by two applications of Lemma 5.2-6. Therefore, the major sub-task remaining is to show that α_i is a prefix of β_i and ω_i is SOE-inclusive of β_i lead to B.

The third and final phase of the proof is to show that A and B imply that the constraints, known to be satisfied by ω_1 and ω_2, must hold for α_1 and α_2. This is a simple deduction, which may be summarized thusly:

Two executions e_1 and e_2 are related in α_1 and α_2 as specified in a constraint

⇒ e_1 and e_2 are so related in ω_1 and ω_2 (by A and B)

⇒ the input/output entries of e_1 and e_2 in ω_1 and ω_2 have the dependency dictated by the constraint

⇒ the input/output entries of e_1 and e_2 in α_1 and α_2 have the same dependency (because every entry in α_i is in ω_i).

Therefore, α_1 and α_2 satisfy the constraint.

The three steps in the proof that all computations in a job satisfy the last five constraints may be found in Sections 5.3.2, 5.3.3, and 5.3.4 respectively. In all of the proofs remaining in this chapter, whenever a program P is given, all initiations, access histories, etc., are with respect to the interpretation Int(P).

5.3.1 Input/Output Types and Pointer Transparency

It is simple to confirm that all canonical computations from $EE(L_{BS},S)$ satisfy the first constraint:

__Lemma 5.3-1__ Let S be any initial standard state for any L_{BS} program P and let Ω be any halted firing sequence starting in S. Then, given $Int(P) = (St,I,IE)$, $\eta(S,\Omega)$ satisfies the Input/Output Type Constraint.

__Proof:__

(1) Let ω be $\eta(S,\Omega)$. Then ω is a computation for $Int(P)$ Lemma 4.3-2

(2) For all d and k, $d\in\{"ID","IT","IF"\} \Rightarrow Ex(d,k)\in IE$ Def. 4.3-2

(3) For all d and k, $d\in DL-\{"ID","IT","IF"\} \Rightarrow Ex(d,k)$ has no output

     entries in ω Def. 4.3-1+Alg. 4.3-1

(4) $\Rightarrow I(d)$ is a pI action   Def. 5.1-2

(5) For all d and k, $d\in DL \Rightarrow$ the input and output entries of $Ex(d,k)$

     are not constrained by the Input/Output Type Constraint

                               (2)+(3)+(4)+Const. 5.1-1

(6) For all d, k, and j, $d\in St-DL \Rightarrow$ the value of the number-j input entry

     to $Ex(d,k)$ in ω is equal to the value of the token removed from d's

     number-j input arc at d's k^{th} firing in Ω Def. 4.3-2+Alg. 4.3-1

(7) For all d, k, and i, let f be any entry such that $T(f)$ has source

     $Src(Ex(d,k),i)$. If $d\in ST-DL$, then there is a prefix $\Delta\varphi$ of Ω con-

     taining exactly k firings of d such that tokens of value $V(f)$

     appear on the number-i output arcs of d at the transition from

     $S\cdot\Delta$ to $S\cdot\Delta\varphi$ Lemma 4.3-1

(8) $\Rightarrow$ tokens of value $V(f)$ are placed on d's number-i output arcs at

     the kth firing of d   Defs. 2.1-5+2.2-5

(9) For any entry f∈ω, f is an input or an output entry of Ex(d,k) where
   d∈St-DL and I(d) is not a structure or a pl action → V(f) is
   not a pointer   (6)+(7)+(8)+Def. 2.2-5

(10) f is an input or an output entry of Ex(d,k) where d∈St-DL and I(d)
    is a structure operation → the type of V(f) depends on I(d) and
    i as in Table 2.2-1   (6)+(7)+Def. 2.2-5

(11) ω satisfies the Input/Output Type Constraint

                              (5)+(9)+(10)+Table 2.2-1+Const. 5.1-1

△

   The proof of the Pointer Transparency Constraint is conceptually
simple but procedurally difficult. The constraint is that for any job J
and computation $\omega_1 \in J$, if ω_2 is any other computation which is identical to
ω_1 to within pointer values, then ω_2 is in J. The proof may be outlined
as follows:

   J is J_E for some equivalence class E of initial states. There is
some $S_1 \in E$ and some halted firing sequence Ω_1 starting in S_1 such that ω_1
is a prefix of some computation β in J_{S_1,Ω_1}. Construct an initial state
S_2, equal to S_1, and a firing sequence Ω_2, equal to Ω_1, such that ω_2 will
be a prefix of some computation in J_{S_2,Ω_2}. Prove that Ω_2 is a halted
firing sequence starting in S_2. Since S_2 equals S_1, S_2 is in E, so ω_2 is
in J_E.

   Proving that Ω_2 is a halted firing sequence starting in S_2, and
verifying that ω_2 is a prefix of $\beta \in J_{S_2,\Omega_2}$, both require the following
fact, first asserted in Section 2.4:

Theorem 5.3-1 For any two equal standard states S_1 and S_2 for the same program P, and any two equal firing sequences Ω_1 starting in S_1 and Ω_2 starting in S_2, $S_2 \cdot \Omega_2$ equals $S_1 \cdot \Omega_1$. Furthermore, if I is the mapping under which the conditions of each arc b in P match in S_1 and S_2, then the mapping under which the conditions of b in $S_1 \cdot \Omega_1$ and $S_2 \cdot \Omega_2$ match is

$$IU\{(n_1,n_2) \mid \exists k: \text{ for } i=1,2, n_i \text{ is the node in the } k^{th} \text{ Copy firing in } \Omega_i\}$$

Proof: (The lengthy proof of this intuitively-correct notion has been deferred to Appendix D.)

$\triangle$

It is next desired to use this result to prove the following: For any two equal states S_1 and S_2 and any two equal firing sequences Ω_1 and Ω_2, if Ω_1 is a halted firing sequence starting in S_1, then Ω_2 is a halted firing sequence starting in S_2. Mere equality of firing sequences says nothing, however, about the pointer-node pairs in their Copy firings. While there is much arbitrariness in choosing these pairs, it is not absolute: Ω_2 is a valid firing sequence starting in $S_2 = (\Gamma, U)$ only if each pointer or node in a pair in a Copy firing in Ω_2 appears in no other pair in a Copy firing or in Π in U (Lemma 5.2-1); i.e., only if the multiset of pointer-node pairs in the Copy firings in Ω_2 is consistent with U. With this added qualification, the assertion can be proven:

Corollary 5.3-1 Let S_1 be any standard state for an L_{BS} program P, and let Ω_1 be any firing sequence starting in S_1. Let S_2 be any standard state equal to S_1, and let Ω_2 be any firing sequence equal to Ω_1. Then

A: Each actor in P is enabled in S_2 iff it is enabled in S_1.

B: If the multiset AP of pointer-node pairs in the Copy firings in Ω_2 is consistent with the heap in S_2, then Ω_2 is a firing sequence starting in S_2, and Ω_2 is halted iff Ω_1 is halted.

Proof: Of A.

(1) There is some one-to-one mapping I under which, for each are b in P, Match$((b,S_2),$ I, $(b,S_1))$ Def. 2.4-3

(2) For each actor d in P, each input and output arc of d has a token in S_2 iff it has a token in S_1 (1)+Def. 2.4-2

(3) Enabling conditions for an actor depend only on the presence or absence of tokens on the actor's input and output arcs   Def. 2.1-4

(4) d is enabled in S_2 iff d is enabled in S_1 (2)+(3)

Prove B by induction on the length of Ω_1.

Basis: $|\Omega_1| = 0$.

(5) $|\Omega_2| = 0$ Def. 2.4-5

(6) Ω_2 is a firing sequence starting in S_2 (5)+Def. 2.3-1

Induction step: Assume the Corollary is true for any Ω_1 of length n, and consider $\Omega_1 = \theta_1\varphi_1$ of length n+1, in which the last firing φ_1 is of the actor labelled d in P.

(7) Ω_2 can be written as $\theta_2\varphi_2$, where φ_2 is also a firing of d and θ_2 equals θ_1 Def. 2.4-5

(8) θ_2 is a firing sequence starting in S_2 ind. hyp.

(9) $S_2 \cdot \theta_2$ equals $S_1 \cdot \theta_1$ (7)+(8)+Thm. 5.3-1

(10) d is enabled in $S_1 \cdot \theta_1$, so d is enabled in $S_2 \cdot \theta_2$ (9)+A+Def. 2.3-1

(11) d is not a Copy operator $\rightarrow \varphi_2 = d$ (7)+Def. 2.3-1

(12) $\rightarrow \theta_2\varphi_2$ is a firing sequence starting in S_2 (8)+(10)+Def. 2.3-1

(13) If d is a Copy operator, then $\varphi_2 = (d,(p,n))$, where p is a pointer

and n is a node   (7)+Def. 2.3-1

(14) $\theta_2\varphi_2$ is not a firing sequence starting in S_2 $\Rightarrow$ (p,n) cannot be

added to Π in going from $S_2 \cdot \theta_2$ to $S_2 \cdot \theta_2 \varphi_2$ (10)+Def. 2.3-1

(15) $\Rightarrow$ p$\notin$dom Π or n$\in$N in $S_2 \cdot \theta_2$ Table 2.2-1

(16) $\Rightarrow$ since no state transition ever diminishes Π or N, either p or n

is in a pair either in Π in S_2 or in some Copy firing in θ_2

(i.e., other than φ_2)   Table 2.2-1

(17) $\Rightarrow$ AP is not consistent with the heap in S_2 (13)+Def. 5.2-3

(18) $\theta_2\varphi_2$ is a firing sequence starting in S_2 (11)+(12)+(14)+(17)

So it is proven by induction that Ω_2 is a firing sequence starting in S_2.

(19) $S_2 \cdot \Omega_2$ equals $S_1 \cdot \Omega_1$ Thm. 5.3-1

(20) Ω_1 is not halted iff there is an actor d in P which is enabled in

$S_1 \cdot \Omega_1$ Def. 2.3-1

(21) iff there is an actor d enabled in $S_2 \cdot \Omega_2$ (19)+A

(22) iff Ω_2 is not halted   Def. 2.3-1

$\triangle$

      The above theorem and corollary express some fundamental properties
of the equality relations between states and between firing sequences.
These will be applied several times in the remainder of the thesis; the
first such application is in the verification of pointer transparency,
as outlined earlier:

Lemma 5.3-2 Let (Int,J) be any expansion from EE(L$_{BS}$,S). Then every job
J$\in J$ satisfies the Pointer Transparency Constraint.

Proof: Let J be any job in J, and let α_1 be any computation in J. Let α_2 be any computation such that

(1) $\alpha_2 \simeq \alpha_1$

(2) There is a total one-to-one mapping Y over V_p such that α_2 can be derived from α_1 by substituting for each entry $f \in \alpha_1$ a similar entry with transfer $T(f)$, and value $V(f)$, if that is not a pointer, or $Y(V(f))$ otherwise   (1)+Def. 5.1-3

(3) (Int, J) is the expansion of some L_{BS} program P, and $J = J_E$ for some equivalence class E of initial standard states for P

                                                    Defs. 4.3-1+4.3-2

(4) There is an initial standard state $S_1 \in E$ and a halted firing sequence Ω_1 starting in S_1 such that α_1 is a prefix of some β_1 in J_{S_1, Ω_1} (3)+Def. 4.3-3

(5) Let $S_1 = (\Gamma_1, U_1)$, where $U_1 = (N_1, \Pi_1, SM_1)$. Let Π_2 be such that $(p,n) \in \Pi_1$ iff $(Y(p),n) \in \Pi_2$. Let $U_2 = (N_1, \Pi_2, SM_1)$. Then since equality of components does not concern pointers, for any $n \in N_1$, $U_2.n \overset{I}{=} U_1.n$, where I is the identity mapping   Def. 2.4-1

(6) Let Γ_2 be Γ_1 with each token which has a pointer value, p, replaced with a token of value $Y(p)$. Then for each arc b in P, either b has no token in both Γ_1 and Γ_2, or b has tokens of identical non-pointer value in Γ_1 and Γ_2, or b has tokens of pointer value v_1 and v_2 in Γ_1 and Γ_2, where $\Pi_2(v_2) = \Pi_1(v_1)$, so $U_2.\Pi_2(v_2) \overset{I}{=} U_1.\Pi_1(v_1)$ (5)

(7) Let $S_2 = (\Gamma_2, U_2)$. Then for each arc b in P,
Match$((b, S_2), I, (b, S_1))$ (6)+Def. 2.4-2

(8) S_2 equals S_1, and so it is in E   (7)+(3)+Def. 2.4-3

(9) Let Ω_2 be Ω_1 with each Copy firing $(d,(p,n))$ replaced by the firing $(d,(Y(p),n))$. Then Ω_2 equals Ω_1 Def. 2.4-5

(10) Let AP be the multiset of pointer-node pairs in the Copy firings in Ω_2. AP is not consistent with the heap in S_2 ⇒ there is a Copy firing $(d,(p,n))$ in Ω_2 such that either p or n is in a pair either in dom Π_2 or in some preceding firing in Ω_2 (7)+(5)+Def. 5.2-3

(11) ⇒ there is a prefix $\theta\varphi$ of Ω_1 in which φ is a Copy firing $(d,(p',n))$ and [n is in a pair in dom Π_2 or in some preceding firing in Ω_2 ⇒ n is in a pair in dom Π_1 or in a firing in θ] and [p is in a pair in dom Π_2 or in some preceding firing in Ω_2 ⇒ $Y^{-1}(p) = p'$ (which is unique since Y is one-to-one) is in a pair in dom Π_1 or in a firing in θ]   (9)+(5)+(2)

(12) ⇒ letting $S_1\cdot\theta$ be (Γ',U') where $U' = (N',\Pi',SM')$, either $p'\in$dom Π' or $n\in N'$ Def. 2.3-1

(13) ⇒ (p',n) could not be added to Π in going from $S_1\cdot\theta$ to $S_1\cdot\theta\varphi$

                    Table 2.2-1

(14) ⇒ Ω_1 is not a firing sequence starting in S_1 (11)+Def. 2.3-1

(15) AP is consistent with the heap in S_2 (4)+(10)+(14)

(16) Ω_2 is a halted firing sequence starting in S_2

                    (4)+(8)+(9)+(10)+(15)+Cor. 5.3-1

(17) Let θ_1 be any prefix of Ω_1 and let θ_2 be the prefix of Ω_2 of the same length. Then θ_2 equals θ_1 (9)+Def. 2.4-5

(18) $S_2\cdot\theta_2$ equals $S_1\cdot\theta_1$, so for each arc b in P, Match$((b,S_2\cdot\theta_2),$ I, $(b,S_1\cdot\theta_1))$, and I is the identity mapping

                    (8)+(17)+(7)+(5)+(9)+Thm. 5.3-1+Def. 2.4-3

(19) Let b be any arc which holds a token in $S_1 \cdot \theta_1$. Then b holds a
token in $S_2 \cdot \theta_2$. For i=1,2, let v_i be the value of the token on
b in $S_i \cdot \theta_i$. Then v_1 is non-pointer $\Rightarrow v_2 = v_1$ (18)+Def. 2.4-2

(20) Assume that v_1, hence v_2, are pointers. Let the heap in $S_i \cdot \theta_i$ be
$U_i = (N_i, \Pi_i, SM_i)$, for $i = 1,2$. Then $U_2 \cdot \Pi_2(v_2) \overset{I}{=} U_1 \cdot \Pi_1(v_1)$

(18)+Def. 2.4-3

(21) $\Pi_2(v_2) = I(\Pi_1(v_1)) = \Pi_1(v_1)$ (20)+(21)+Def. 2.4-1

(22) v_1 is in dom Π_1 (19)+Thm. 2.2-1

(23) v_1 is in dom Π in $S_1 \Rightarrow v_2 = Y(v_1)$ (4)+(21)

(24) v_1 is not in dom Π in $S_1 \Rightarrow$ there is a Copy firing in θ_1 containing
the ordered pair $(v_1, \Pi_1(v_1))$ (22)+Def. 2.3-1

(25) $\Rightarrow$ there is a Copy firing in θ_2 containing $(Y(v_1), \Pi_1(v_1))$, which
equals $(Y(v_1), \Pi_2(v_2))$ (9)+(17)+(21)

(26) Since $\Pi_2(v_2)$ is in at most one pair in Ω_2, that Copy firing contains
the pair $(v_2, \Pi_2(v_2))$ (15)+Defs. 5.2-3+2.3-1

(27) v_1 is not in dom Π in $S_1 \Rightarrow v_2 = Y(v_1)$ (24)+(25)+(26)

(28) v_1 is a pointer $\Rightarrow v_2 = Y(v_1)$ (23)+(27)

(29) For any m, and for i=1,2, let $\Delta_i\varphi_i$ be the length-m prefix of Ω_i.
Then φ_1 and φ_2 are firings of the same actor d and $|\Delta_1| = |\Delta_2|$.
Then if d is a merge gate, its control input arc holds tokens of
the same value in $S_1 \cdot \Delta_1$ and $S_2 \cdot \Delta_2$ (17)+(19)

(30) φ_1 and φ_2 remove tokens from the same set of arcs

(29)+Defs. 2.1-5+2.2-5

(31) The token on b in $S_1 \cdot \theta_1$ was on b in S_1 iff there was a token on b
in S_1 and there is no prefix $\Delta_1\varphi_1$ of θ_1 in which φ_1 removes a token
from b iff there is a token on b in S_2 and there is no prefix $\Delta_2\varphi_2$

of θ_2 in which φ_2 removes a token from b iff the token on b in

$S_2 \cdot \theta_2$ was on b in S_2 (29)+(30)+Defs. 2.4-3+2.4-2

(32) For each actor d', there are n' firings of d' in θ_1 iff there are

n' firings of d' in θ_2 (17)+Def. 2.4-5

(33) Source(b,S_2,θ_2) = Source(b,S_1,θ_i)   (31)+(32)+Alg. 4.3-1

(34) There is an entry f in $\omega(S_1,\Omega_1)$ with V(f) = v_1 and transfer

T(f) = (s,Dst(Ex(d,k),j)) iff there is a prefix $\theta_1\varphi_1$ of Ω_1 in which

φ_1 is the kth firing of the actor labelled d, that firing removes

a token of value v_1 from d's number-j input arc b, and

s = Source(b,S_1,θ_1)   Alg. 4.3-1

(35) iff there is a prefix $\theta_2\varphi_2$ of Ω_2 (of the same length as $\theta_1\varphi_1$) in

which φ_2 is the kth firing of d   (9)

(36) and that firing removes a token from b   (29)+(30)

(37) and that token has value v_2 where

$$v_2 = \begin{cases} v_1 & \text{if } v_1 \text{ is not a pointer} \\ Y(v_1) & \text{otherwise} \end{cases}$$ (17)+(19)+(28)

(38) and s = Source(b,S_2,θ_2)   (17)+(33)

(39) iff there is an entry g in $\omega(S_2,\Omega_2)$ with V(g) = v_2 and T(g) = T(f)

Alg. 4.3-1

(40) There is an entry f in $\eta(S_1,\Omega_1)$ with V(f) = v_1 and transfer

T(f) = (s,Dst(Ex(d,k),j)) iff there is such an entry in $\omega(S_1,\Omega_1)$

or there is an arc b in P which holds a token in $S_1 \cdot \Omega_1$ of value v_1,

d, k, and j are related to b as in Alg. 4.3-1, and

s = Source(b,S_1,Ω_1)   Alg. 4.3-1

(41) iff there is such an entry in $\omega(S_2,\Omega_2)$ or there is an arc b which

holds in $S_2 \cdot \Omega_2$ a token of value v_2 where

$$v_2 = \begin{cases} v_1 & \text{if } v_1 \text{ is non-pointer} \\ Y(v_1) & \text{otherwise} \end{cases}$$
(34)+(38)+(17)+(9)+(19)+(28)

(42) and d, k, and j are related to b as in Alg. 4.3-1 and

$s = \text{Source}(b, S_2, \Omega_2)$ (17)+(9)+(33)

(43) iff there is an entry g in $\eta(S_2, \Omega_2)$ with $V(g) = v_2$ and $T(g) = T(f)$

Alg. 4.3-1

(44) β_1 is a permutation of $\eta(S_1, \Omega_1)$ which is causal wrt Int(P), and

$\Phi(\beta_1)$ is the reduction of Ω_1 (4)+Def. 4.3-5

(45) There is a permutation β_2 of $\eta(S_2, \Omega_2)$ which can be derived from β_1

by replacing each entry f in β_1 with an entry g such that

$T(g) = T(f)$ and $V(g)$ is $V(f)$, if that is not a pointer, or $Y(V(f))$

otherwise (44)+(40)+(43)

(46) The prefix of β_2 of length $|\alpha_1|$ is α_2 (45)+(2)+(4)

(47) Let f be any entry in β_1, and let g be the entry in β_2 with

$T(g) = T(f)$. Then f is the initiating entry wrt Int(P) of an

execution e iff g is (45)+Def. 4.2-6

(48) $\Phi(\beta_2) = \Phi(\beta_1)$ (47)+Def. 4.3-4

(49) $\Phi(\beta_2)$ is the reduction of Ω_1, which is the reduction of Ω_2

(48)+(44)+(9)+Def. 2.4-5

(50) Let $\gamma_2 g$ be any prefix of β_2. Then g is an output entry of e $\Leftrightarrow$ for

the same-length prefix $\gamma_1 f$ of β_1, f is an output entry of e

(45)+Def. 4.2-5

(51) $\Leftrightarrow$ e's initiating entry wrt Int(P) is in γ_1 (44)+Def. 4.2-7

(52) $\Leftrightarrow$ e's initiating entry is in γ_2 (47)

(53) β_2 is causal with respect to $Int(P)$ (50)+(52)+Def. 5.2-7

(54) Let $\gamma_2 g$ be any prefix of β_2, and let θ_2 be the prefix of Ω_2 whose

     reduction is $\Phi(\gamma_2)$. Let $\gamma_1 f$ be the same-length prefix of Ω_1.

     Then $T(f) = T(g)$ and $\Phi(\gamma_1) = \Phi(\gamma_2)$ (45)+(47)+Def. 4.3-4

(55) Let θ_1 be the prefix of Ω_1 whose reduction is $\Phi(\gamma_1)$. Then θ_1 equals

     θ_2 and $|\theta_2| = |\theta_1|$ (54)+Def. 2.4-5

(56) $T(g)$ has destination $Dst(Ex(d,k),j)$ and $d\notin DL \Rightarrow T(f)$ has the same

     destination and $d\notin DL \Rightarrow d$ is enabled in $S_1 \cdot \theta_1$ and if d is a merge

     gate and its number-j input arc b is its T (F) input arc, then d's

     control input arc has a <u>true</u> (<u>false</u>) token in $S_1 \cdot \theta_1$

                                (54)+(55)+(4)+Def. 2.4-5

(57) $\Rightarrow d$ is enabled in $S_2 \cdot \theta_2$ and if d is a merge gate and b is its T (F)

     input arc, then its control input arc holds a <u>true</u> (<u>false</u>) token

     in $S_2 \cdot \theta_2$ (8)+(55)+(29)+Cor. 5.3-1

(58) For any firing sequence Δ_2 starting in $S_2 \cdot \theta_2$, it is possible to

     change the pointer-node pairs in the Copy firings in Δ_2 to derive

     an equal firing sequence which is consistent with the heap in

     $S_1 \cdot \theta_1$ Defs. 2.4-5+5.2-3

(59) For any firing sequence Δ_2 starting in $S_2 \cdot \theta_2$, there is an equal

     firing sequence Δ_1 starting in $S_1 \cdot \theta_1$ (58)+Cor. 5.3-1

(60) $T(g)$ has destination $Dst(Ex(d,k),j)$, $d\in DL$ and $d=(c,n) \Rightarrow T(f)$ has

     the same destination $\Rightarrow$ letting b be the number-n program output

     arc of P, if $c =$ "OD", or else the number-n input arc of c,

     there is a token on b in $S_1 \cdot \theta_1$ and if c is an actor label, there

     is no firing sequence starting in $S_1 \cdot \theta_1$ which contains a firing

     of c (54)+(55)+(4)+Def. 4.3-5

(61) $\rightarrow$ there is a token on b in $S_2 \cdot \theta_2$ and if c is an actor label, there

is no firing sequence starting in $S_2 \cdot \theta_2$ which contains a firing

of c $\qquad$ (55)+(17)+(19)+(59)+Def. 2.4-5

(62) β_2 is in J_{S_2, Ω_2} $\quad$ (45)+(49)+(53)+(54)+(56)+(57)+(60)+(61)+Def. 2.4-5

(63) α_2 is in J $\qquad$ (8)+(16)+(62)+(46)

$\triangle$

This essentially completes the confirmation that the first two S-S

constraints are satisfied in $EE(L_{BS}, S)$. The next subsection commences

the proof for the final five constraints.

5.3.2 Canonical Computations

The purpose here is to demonstrate that the remaining constraints

are satisfied by any canonical computation or pair of canonical compu-

tations in a job; this is the first step in proving that they are

satisfied by any computations.

The Atomic Output and Structure Output Constraints concern the output

entries of an execution e_1 which is in the reach of another execution e_2.

The proof, an eloquent testimonial to the utility of the definition of the

heap determined by a computation, has already been outlined at the start

of Section 5.2.

Lemma 5.3-3 Let S be any initial standard state for an L_{BS} program P, and

let Ω be any halted firing sequence starting in S. Then $\eta(S, \Omega)$ satisifes

the Atomic Output Constraint and the Structure Output Constraint (given

Int(P)).

Proof:

(1) Let $\omega = \eta(S,\Omega)$ and let f be any entry in ω such that T(f) has

source Src(e,i) for any Fetch, Assign, Select, Update, or Delete

execution e = Ex(d,k) and any i. Let Int(P) be (St,l,IE). Then

d∈St-DL   Defs. 4.3-2+4.3-1

(2) There is a prefix $\theta\varphi$ of Ω containing exactly k firings of d such

that tokens of value V(f) appear on the number-1 group of output

arcs of d at the transition from $S\cdot\theta$ to $S\cdot\theta\varphi$ (1)+Lemma 4.3-1

(3) φ must be the kth firing of d in Ω (2)+Defs. 2.1-5+2.2-5

(4) Let the heap in $S\cdot\theta$ be (N,Π,SM). Let α be $\eta(S,\theta)$ and let NAR be

the node activation record derived from θ and α. Then the heap

determined by α from the heap in S and NAR, (N$_\alpha$,Π_α,SM$_\alpha$), is

defined and is identical to (N,Π,SM)   Thm. 5.2-1

(5) Let β be $\eta(S,\theta\varphi)$, let p be the value of the number-1 input to φ,

and let n = Π(p). Then g = Ent$_\beta$(e,1) is in β but not in α, V(g)

is p, and there are m input entries to e in β, where m tokens are

removed by φ (3)+(1)+Alg. 4.3-1

(6) V(g) = p is a pointer   (5)+(1)+Def. 2.2-5

(7) m = In(l(d)), so e is initiated in β (5)+Defs. 4.3-2+4.3-1+4.2-6

(8) β is a prefix of ω, as is α (5)+(2)+Alg. 4.3-1

(9) β and ω are both causal computations for Int(P)   Lemma 4.3-2

(10) For any pointer q, q is the value of the output entries in ω of a

Copy execution C $\Rightarrow$ the first entry in ω with value q is an output

entry of C   Lemma 5.2-3

(11) For any Assign, Update, or Delete execution e', e∈R(e') in ω $\Rightarrow$

e∈R(e') in β (7)-(10)+Lemma 5.2-6

(12) $\Rightarrow$ g is in duration $D(e')$ in β (5)+Defs. 5.1-6+5.1-8

(13) $\Rightarrow D(e')$ extends to the end of H_p^{α} (2)+(4)+(5)+(6)+Lemma 5.2-7

(14) $p \in \text{dom } \Pi = \text{dom } \Pi_{\alpha}$ (5)+Thm. 2.2-1

(15) $e \in R(A)$ for Assign execution $A \Rightarrow D(A)$ extends to the end of H_p^{α}

                                                       (11)+(13)

(16) $\Rightarrow$ the value in $SM_{\alpha}(n) = SM(n)$ is $v = V(\text{Ent}_{\alpha}(A,2))$ (14)+(4)+Def. 5.2-7

(17) $\Rightarrow$ if φ is a Fetch firing $\wedge$ i = 1, then the value placed on the

number-i group of output arcs of d by φ is v   (2)-(5)+Table 2.2-1

(18) $\Rightarrow$ if e a Fetch execution $\wedge$ i = 1, then $V(f) = v$ (1)+(2)+(3)

(19) $e \in R(A)$ for Assign execution A, i = 2, and φ is a Fetch or Assign

firing $\Rightarrow$ the value placed on the number-i group of output arcs of

d by φ is the value of the predicate (v$\neq$<u>nil</u>)

                                           (15)+(16)+(2)-(5)+Table 2.2-1

(20) $\Rightarrow$ e is a Fetch or Assign execution, i = 2, and $V(f) = $ (v$\neq$<u>nil</u>)

                                               (1)+(2)+(3)

(21) ω satisfies the Atomic Output Constraint

                        (1)+(16)+(18)+(20)+Def. 4.2-6+Table 2.2-1

(22) Assume e is a Select, Update, or Delete execution. Then φ is a

Select, Update, or Delete firing, and the selector input s to φ

equals $V(\text{Ent}_{\omega}(e,2))$ (1)+(3)+Alg. 4.3-1

(23) $e \in R(D)$ for Delete execution $D \Rightarrow V(\text{Ent}_{\omega}(D,2)) = s$ (22)+Def. 5.1-8

(24) $\Rightarrow$ there is no ordered pair containing s in $SM(n)$

                                 (11)+(13)+(14)+(4)+Def. 5.2-7

(25) $\Rightarrow$ if e is a Select execution $\wedge$ i = 1, then $V(f) = $ <u>undef</u>, and if

i = 2, then $V(f) = $ <u>false</u>   (1)-(5)+Table 2.2-1

(26) $e \in R(U)$ for Update execution $U \Rightarrow$ the ordered pair $(s, \Pi(r))$ is in

   $SM(n)$, where $r = V(Ent_\alpha(U,3))$

$$(22)+(11)+(13)+(14)+(4)+Defs.\ 5.1-8+5.2-7$$

(27) $\Rightarrow$ if e is a Select execution $\wedge$ $i = 1$, then $V(f) = V(Ent_\omega(U,3))$,

   and if $i = 2$, then $V(f) = \underline{true}$ $(1)-(5)+(8)+Table\ 2.2-1$

(28) ω satisfies the Structure Output Constraint

$$(1)+(23)+(25)+(26)+(27)+Def.\ 4.2-6+Const.\ 5.1-4$$

$\triangle$

Not every execution in a computation falls into a reach; further-more, even if a First or Next execution e does fall into one or more reaches, these do not completely determine the set of selectors upon which e's output entries depend. Under a certain condition, however, the output entries of two such executions e_1 and e_2 must have the same value in two computations ω_1 and ω_2 in the same job: if the pointer inputs p_1 and p_2 to e_1 and e_2 are such that $(p_1, \omega_1) \rho (p_2, \omega_2)$. This assertion is expressed in the Initial Structure and the First/Next Output Constraints.

The proof that any pair of canonical computations ω_1 and ω_2 in a job satisfies these two constraints proceeds along the following lines:

1. For $i=1,2$, $\omega_i = \eta(S_i, \Omega_i)$, where S_1 and S_2 are two initial states which are equal under some mapping I.

2. The output entries of e_i can be related to the content of a partic-ular node in the heap $U_i = (N_i, \Pi_i, SM_i)$ in S_i. The relationship is derived from the heap determined by ω_i from U_i, by applying reason-ing similar to that just used in Lemma 5.3-3. The particular node in N_i is $\Pi_i(q_i)$, where q_i is the unique pointer in dom Π_i such that $DD_{\omega_i}(q_i, p_i)$ (Lemma 5.3-4 below).

3. $(p_1, \omega_1) \rho (p_2, \omega_2) \to (q_1, \omega_1) \rho (q_2, \omega_2) \to q_1$ and q_2 point to equal components (under I) of U_1 and U_2; thus the contents of $\Pi_1(q_1)$ and $\Pi_2(q_2)$ have identical values and selector sets (Theorem 5.3-2).

4. The non-pointer output entries of e_1 and e_2 are identical (Lemma 5.3-5).

Lemma 5.3-4 Let S be any initial state for an L_{BS} program P, and let the heap in S be (N, Π, SM). Let ω be any firing sequence starting in S, let ω be $\eta(S, \omega)$, and let e be any execution of any structure operator (except Copy). Let p be $V(Ent(e,1))$, let q be the unique pointer in dom Π such that $DD_\omega(q,p)$, and let $n = \Pi(q)$. Then the conclusions depicted in Table 5.3-1 can be drawn about the values of e's output entries in ω.

Proof: (The reasoning here is so similar to that in Lemma 5.3-3 that the proof has been removed to Appendix D.)

Theorem 5.3-2 Let S_1 and S_2 be any two equal initial standard states for the same L_{BS} program P. Let I be the single one-to-one mapping under which the conditions in S_1 and S_2 of each arc in P match. For $i=1,2$, let the heap in S_i be $U_i = (N_i, \Pi_i, SM_i)$, let ω_i be any firing sequence starting in S_i, and let ω_i be $\eta(S_i, \omega_i)$. Let ρ be the equal pointer relation defined from Int(P). Assuming that ω_1 and ω_2 are both computations for Int(P), for any two pointers p_1 and p_2,

A: $(p_1, \omega_1) \rho (p_2, \omega_2) \to (q_1, \omega_1) \rho (q_2, \omega_2)$, where, for $i=1,2$, q_i is the unique pointer in dom Π_i such that $DD_{\omega_i}(q_i, p_i)$.

B: $p_1 \in \text{dom } \Pi_1$, $p_2 \in \text{dom } \Pi_2$, and $(p_1, \omega_1) \rho (p_2, \omega_2) \to$

If e is a Fetch or Assign execution and is not in a reach, then the value of $Src(e,i)$ is given below, where v is the value in $SM(n)$.

Type of e	i = 1	i = 2
Fetch	v	v≠nil
Assign	0	v≠nil

If e is a Select, Update, or Delete execution and is not in a reach, then the value of $Src(e,i)$ is given below, where $s = V(Ent_{\omega}(e,2))$.

If there is an r such that $(s,\Pi(r)) \in SM(n)$:

Type of e	i = 1	i = 2
Select	r	true
Update/Delete	0	true

Otherwise:

Type of e	i = 1	i = 2
Select	undef	false
Update/Delete	0	false

If e is a First execution or a Next execution with selector input s, then the value of $Src(e,i)$ depends just on s and the set S of selectors defined by:

$S = (S^a - S^b) \cup S^c$, where

$S^a = \{s \in \Sigma \mid \exists m: (s,m) \in SM(n)\}$,

$S^b = \{s \in \Sigma \mid \exists Delete\ D: e \in R(D)\ in\ \omega\ and\ s = V(Ent_{\omega}(D,2))\}$,

and $S^c = \{s \in \Sigma \mid \exists Update\ U: e \in R(U)\ in\ \omega\ and\ s = V(Ent_{\omega}(U,2))\}$

Outputs of an Execution which is Not in a Reach

Table 5.3-1

1. There is an arc of P which, for i=1,2, holds a pointer r_i in S_i
    such that $\Pi_1(p_1)$ equals or is reachable from $\Pi_1(r_i)$ in U_1.

2. $\Pi_2(p_2) = I(\Pi_1(p_1))$

3. $SM_2(\Pi_2(p_2)) = I(SM_1(\Pi_1(p_1)))$

Proof:

(1) Let $Int(P) = (ST, I, IE)$. Then $(p_1,\omega_1)\rho(p_2,\omega_2) \Rightarrow$

(1a) There is a source $s = Src(e,i)$ for some $e\in IE$ and some i such that
    p_1 (p_2) is the value of s in ω_1 (ω_2), or

(1b) There are Select executions S_1 and S_2 such that

    p_i is the value of $Src(S_i,1)$ in ω_1, i=1,2,

    S_i does not fall into a reach in ω_i, i=1,2,

    $V(Ent_{\omega_1}(S_1,2)) = V(Ent_{\omega_2}(S_2,2))$, and

    $(V(Ent_{\omega_1}(S_1,1)),\omega_1)\rho(V(Ent_{\omega_2}(S_2,1)),\omega_2)$, or

(1c) $\exists q \neq p_1$ such that $DD_{\omega_1}(q,p_1)$ and $(q,\omega_1)\rho(p_2,\omega_2)$ Def. 5.1-10

Proof is by induction on the smallest number n of recursive applications

of the above three rules required to derive that $(p_1,\omega_1)\rho(p_2,\omega_2)$. Induc-

tion hypotheses are A with the addition of "The shortest derivation of

$(q_1,\omega_1)\rho(q_2,\omega_2)$ has no more steps than the shortest derivation of

$(p_1,\omega_1)\rho(p_2,\omega_2)$," and B.

(2) (1a) is true of p_1 and p_2 $\Rightarrow$ there is a one-step derivation, so n=1

(3) The last step in the shortest derivation is an application of (1b)
    or (1c) $\Rightarrow$ there is a pair of pointers q_1 and q_2, not the same as p_1
    and p_2, such that it has been derived that $(q_1,\omega_1)\rho(q_2,\omega_2)$

(4) $\Rightarrow$ there is at least one additional step in the derivation, so $n > 1$

Basis: n = 1.

(5) (1a) is true of p_1 and p_2 (1)+(3)+(4)

(6) $e \in \{Ex(ID,0), Ex(IT,0), Ex(IF,0)\}$ (5)+(1a)+Def. 4.3-2

(7) There is a pointer of value p_1 (p_2) on the number-i program input

arc of P in S_1 (S_2)   (6)+Alg. 4.3-1

(8) There is an arc b in P which has a token of value p_1 (p_2) in S_1

(S_2) and $p_1 \in \text{dom } \Pi_1$ ($p_2 \in \text{dom } \Pi_2$)   (7)+Def. 2.2-6

(9) For i=1,2, $p_i = q_i$, the unique pointer in dom Π_i such that

$DD_{\omega_i}(q_i, p_i)$ (8)+Def. 5.1-9

(10) $(q_1, \omega_1) \rho (q_2, \omega_2)$, and the shortest derivation of this has no more

steps than the shortest derivation of $(p_1, \omega_1) \rho (p_2, \omega_2)$ (1)+(9)

(11) $\text{Match}((b, S_1), I, (b, S_2))$ (8)+Def. 2.4-3

(12) $U_2 . \Pi_2(p_2) \overset{I}{=} U_1 . \Pi_1(p_1)$ (8)+(11)+Def. 2.4-2

(13) $\Pi_2(p_2) = I(\Pi_1(p_1))$ and $SM_2(\Pi_2(p_2)) = I(SM_1(\Pi_1(p_1)))$ (12)+Def. 2.4-1

Induction step: Assume that the induction hypotheses are true for any

p_1 and p_2 if the shortest derivation of $(p_1, \omega_1) \rho (p_2, \omega_2)$ has n or fewer

steps, n > 0. Consider

(14) p_1 and p_2 for which the shortest derivation has n+1 steps

(15) Either (1b) or (1c) is applied as the last step   (1)+(2)

(16) (1b) is the last step; i.e., is true of p_1 and $p_2 \Rightarrow$ since S_1 is not

in a reach in ω_1, letting p_i' be $V(\text{Ent}_{\omega_i}(S_i, 1))$ and s be

$V(\text{Ent}_{\omega_i}(S_i, 2))$, the pair $(s, \Pi_i(p_i))$ is in $SM_i(\Pi_i(q_i'))$ where q_i' is

the unique pointer in dom Π_i such that $DD_{\omega_i}(q_i', p_i)$ (1b)+Lemma 5.3-4

(17) $\Rightarrow p_i \in \text{dom } \Pi_i$ Defs. 2.2-6+2.2-1

(18) $\Rightarrow (q_1, \omega_1) \rho (q_2, \omega_2)$ and the shortest derivation of this has the same

number of steps as the shortest derivation of $(p_1, \omega_1) \rho (p_2, \omega_2)$

(8)-(10)

(19) (1c) is the last step $\Rightarrow \exists p_1' \neq p_1$: $DD_{\omega_1}(p_1',p_1)$ and $(p_1',\omega_1)\rho(p_2,\omega_2) \Rightarrow$

the shortest derivation of $(p_1',\omega_1)\rho(p_2,\omega_2)$ has n steps $\hspace{2em}$ (14)

(20) $\Rightarrow (q_1,\omega_1)\rho(q_2,\omega_2)$, where q_1 is the unique pointer in dom Π_1 such

that $DD_{\omega_1}(q_1,p_1')$ and $DD_{\omega_2}(q_2,p_2)$, and the shortest derivation of

this has no more than n steps $\hspace{4em}$ ind. hyp. A

(21) $\Rightarrow DD_{\omega_1}(q_1,p_1)$ $\hspace{6em}$ (19)+Def. 5.1-9

(22) $\Rightarrow (q_1,\omega_1)\rho(q_2,\omega_2)$, where q_1 is the unique pointer in dom Π_1 such

that $DD_{\omega_1}(q_1,p_1)$, and the shortest derivation of this has no more

steps than the shortest derivation of $(p_1,\omega_1)\rho(p_2,\omega_2)$ $\hspace{1em}$ (20)+(14)

(23) A for p_1 and p_2 $\hspace{4em}$ (15)+(16)+(18)+(19)+(22)

(24) (1c) is true of p_1 and $p_2 \Rightarrow \exists p_1' \neq p_1$: $DD_{\omega_1}(p_1',p_1) \Rightarrow p_1$ is the value

of the output entries of a Copy execution in ω_1 $\hspace{2em}$ Def. 5.1-9

(25) $\Rightarrow p_1 \notin$ dom $\Pi_1 \Rightarrow$ B is vacuously true $\hspace{2em}$ Lemma 5.2-3

(26) (1b) is the last step applied $\Rightarrow$ letting p_1' be $V(Ent_{\omega_1}(S_1,1))$ and s

be $V(Ent_{\omega_1}(S_1,2))$, the pair $(s,\Pi_1(p_1))$ is in $SM_1(\Pi_1(q_1'))$, where

q_1' is the unique pointer in dom Π_1 such that $DD_{\omega_1}(q_1',p_1')$ and

$(p_1',\omega_1)\rho(p_2',\omega_2)$ $\hspace{6em}$ (16)+(1b)

(27) $\Rightarrow$ the shortest derivation of $(p_1,\omega_1)\rho(p_2,\omega_2)$ consists of an appli-

cation of (1b) following the shortest derivation of

$(p_1',\omega_1)\rho(p_2',\omega_2)$, which has n steps $\hspace{2em}$ (1)+(14)

(28) $\Rightarrow (q_1',\omega_1)\rho(q_2',\omega_2)$ and the shortest derivation of this contains no

more steps than the shortest derivation of $(p_1',\omega_1)\rho(p_2',\omega_2)$

$\hspace{22em}$ (26)+ind. hyp. A

(29) $\Rightarrow$ the shortest derivation of $(q_1',\omega_1)\rho(q_2',\omega_2)$ has n or fewer steps

$\hspace{24em}$ (27)

(30) $\Rightarrow SM_2(\Pi_2(q_2')) = I(SM_1(\Pi_1(q_1')))$, and there is an arc of P which holds

a pointer r_1 in S_1 such that $\Pi_1(q_1')$ equals or is reachable from

$\Pi_1(r_1)$ in U_1 (26)+ind. hyp. B

(31) $\Rightarrow$ since $(s,\Pi_1(p_1)) \in SM_1(\Pi_1(q_1'))$, $\Pi_2(p_2) = I(\Pi_1(p_1))$ and $\Pi_1(p_1)$ is a

successor of $\Pi_1(q_1')$ (26)+Defs. 2.4-1+2.2-2

(32) $\wedge\ U_2.\Pi_2(r_2) \overset{I}{=} U_1.\Pi_1(r_1)$ Defs. 2.4-3+2.4-2

(33) $\Rightarrow \Pi_1(p_1)$ is reachable from $\Pi_1(r_1)$ (30)+Def. 2.2-2

(34) $\Rightarrow SM_2(I(\Pi_1(p_1))) = I(SM_1(\Pi_1(p_1)))$ (32)+Def. 2.4-1

(35) $\Rightarrow SM_2(\Pi_2(p_2)) = I(SM_1(\Pi_1(p_1)))$ (31)

(36) B for p_1 and p_2 (15)+(24)+(25)+(26)+(30)+(31)+(35)

$\triangle$

__Lemma 5.3-5__ For any L_{BS} program P, let S_1 and S_2 be any two equal initial standard states for P. For i=1,2, let Ω_i be any halted firing sequence starting in S_i and let $\omega_i = \eta(S_i,\Omega_i)$. Then, given Int(P), the pair consisting of ω_1 and ω_2 satisfies the Initial Structure Constraint and the First/Next Output Constraint.

Proof: (The proof of this is simply a detailed expansion of the outline given on page 259, and so has been deferred to Appendix D.)

$\triangle$

The final constraint is the Unique Pointer Generation Constraint. Briefly, this states that the pointer output of a Copy execution in a computation ω must be different from the pointer output of any input execution (that is, one in IE), any other Copy execution, and any Select execution which does not fall into a reach in ω.

__Lemma 5.3-6__ For any initial standard state S for any L_{BS} program P, and for any halted firing sequence Ω starting in S, $\eta(S,\Omega)$ satisfies the Unique Pointer Generation Constraint.

Proof:

(1) Let $\omega = \eta(S, \Omega)$. Then ω is a computation for $Int(P) = (St, l, IE)$

Lemma 4.3-2

(2) Let p be the value in ω of the output entries of any Copy execution

    C. Let the heap in S be (N, Π, SM). Then $p \notin dom \, \Pi$ and no other

    Copy execution has output entries in ω of value p   (1)+Lemma 5.2-3

(3) For any pointer p', p' is the value of the output entries of any

    execution $e \in IE \Rightarrow e \in \{Ex(ID,0), Ex(IT,0), Ex(IF,0)\}$ (1)+Def. 4.3-2

(4) $\Rightarrow$ p' is on an arc of p in S Alg. 4.3-1

(5) $\Rightarrow$ p' is in dom $\Pi \Rightarrow p' \neq p$ (2)+Def. 2.2-6

(6) For any pointer p', p' is the value of the output entries in ω of a

    Select execution S which does not fall into a reach $\Rightarrow (s, \Pi(p'))$

    is in $SM(\Pi(q))$ for some $q \in dom \, \Pi$ (2)+(1)+Lemma 5.3-4

(7) $\Rightarrow \Pi(p') \in N \Rightarrow p' \in dom \, \Pi \Rightarrow p' \neq p$ (2)+Def. 2.2-1

(8) ω satisfies the Unique Pointer Generation Constraint

(2)+(3)+(5)+(6)+(7)+Const. 5.1-7

$\triangle$

5.3.3 The Qualifying Relationships

This subsection provides the results necessary to complete the second step of the proof that all computations in a job satisfy the final five constraints. Recalling the comprehensive outline provided at the start of Section 5.3, the goal here is to prove that for any two computations α_1 and α_2 in a job J, there are canonical computations ω_1 and ω_2 in J such that:

A: For i=1,2, for any execution e_1 initiated in α_1, e_1 is in $R(e_2)$ in α_1 iff e_1 is in $R(e_2)$ in ω_1.

B: For any two pointers p_1 and p_2, $(p_1, \alpha_1)\rho(p_2, \alpha_2) \Rightarrow (p_1, \omega_1)\rho(p_2, \omega_2)$.

For $i=1,2$, the appropriate ω_i is found from α_i as follows: Every α is a prefix of some β in $J_{S,\Omega}$, where S is an initial standard state and Ω is a halted firing sequence starting in S; in turn β is a permutation of $\omega = \eta(S,\Omega)$.

A follows from two invocations of an earlier, general result relating reaches in a pair of computations (Lemma 5.2-6). Two requirements must be met by any pair γ and δ before this result can be applied to them. First is that either γ is a prefix of δ or δ is SOE-inclusive of γ. The β_i selected above has α_i as a prefix, and it is easily confirmed that ω_i is SOE-inclusive of β_i (Lemma 5.3-7 below). The second requirement is that

> For every pointer p, p is the value in δ of the output entries of a
> Copy execution C $\Rightarrow$ the first entry in δ with value p is an output
> entry of C.

This has already been established for all canonical computations δ, such as ω_i (Lemma 5.2-3). It is here proven for $\delta = \beta_i$ by an indirect, two-step process. First it is shown to be true for any causal computation which satisfies the Input/Output Type, Structure Output, and Unique Pointer Generation Constraints (Lemma 5.3-8). Then it is shown that, since ω_i is known to satisfy these constraints, any computation of which it is SOE-inclusive, including β_i, must satisfy them as well (Lemma 5.3-9). (The reason for this two-part development is that the first lemma is in a form which can be used several times in Chapter 6.) With these preliminaries, Lemma 5.2-6 can be applied first to α_i and β_i, then to β_i and ω_i, to yield A.

Finally, Lemma 5.3-10 displays the simple manipulations of definitions needed to prove that A implies B.

Lemma 5.3-7 Let S be any (standard or modified) initial state for an L_{BS} program P, and let Ω be any halted firing sequence starting in S. Let ω be $\eta(S,\Omega)$ and let β be any computation in $J_{S,\Omega}$. Then ω is SOE-inclusive of β.

Proof:

(1) β is a causal permutation of ω such that $\Phi(\beta)$ is the reduction of Ω

Def. 4.3-5

(2) ω is also in $J_{S,\Omega}$, and $\Phi(\omega)$ is the reduction of Ω Lemma 4.3-3

(3) ω is a computation for Int(P), so β is as well

(1)+Lemma 4.3-2+Def. 4.2-6

(4) Let Int(P) be (St,I,IE), and let $e = Ex(d,k)$ be any execution in which $I(d)$ is a structure operation. Then $d \in St-DL$ Def. 4.3-2

(5) e is initiated in β $\Rightarrow$ there are $In(I(d))$ input entries to e in β $\Rightarrow$ there are $In(I(d))$ input entries to e in ω $\Rightarrow$ e is initiated in ω

(1)+Def. 4.2-6

(6) Let NDE be the set of executions $NDE = \{Ex(d,k) \mid d \in St-DL\}$. For any $Ex(d,k)$ in NDE which is initiated in β, the initiating entry to e is preceded in both β and ω by the initiating entry to exactly $k-1$ other executions of d   (4)+(2)+(5)+Cor. 4.3-1

(7) For any $n \leq |\Phi(\beta)| = |\Phi(\omega)|$, the nth execution in NDE to initiate in β is $Ex(d,k)$ iff the nth firing in $\Phi(\beta)$ is a firing of d and is preceded by exactly $k-1$ other firings of d; i.e., is the kth firing of d   (1)+(2)+(6)+Def. 4.3-4

(8) iff the nth firing in $\Phi(\omega)$ is the kth firing of d   (1)+(2)

(9) iff the nth execution in NDE to initiate in ω is $Ex(d,k)$ Def. 4.3-4

(10) Let e and e' be any two distinct structure operation executions
such that e is initiated in β. Then both e and e' are in NDE

$$(4)+(6)$$

(11) e' initiates before e in β iff e is the nth execution in NDE to
initiate in β, $n \leq |\Phi(\beta)|$, e' is the mth, and $m < n$ iff e is the
n^{th} execution in NDE to initiate in ω, $n \leq |\Phi(\omega)|$, e' is the mth,
and $m < n$ iff e' initiates before e in ω $\qquad$ $(10)+(7)+(9)$

(12) Any execution which has output entries in ω has output entries in β.
For every entry $f \in \beta$, there is an entry with the same value in ω
whose transfer has the same source as $T(f)$. For any execution e
and for any j, if there is an entry $Ent_\beta(e,j)$ in β, then there is
an entry $Ent_\omega(e,j)$ in ω with the same value $\qquad$ (1)

(13) ω is SOE-inclusive of β $\qquad$ $(3)+(4)+(5)+(10)+(11)+(12)+$Def. 5.2-8

$$\triangle$$

Lemma 5.3-8 Let ω be any causal computation for interpretation (St, I, IE)
which satisfies the Input/Output Type, Structure Output, and Unique
Pointer Generation Constraints. Then:

A: For any pointer p which is the value of an entry in ω, the first
entry in ω with value p is an output entry of an execution which
either is in IE, is a Copy execution, or is a Select execution
which is in no reach in ω.

B: For any pointer p which is the value in ω of the output entries of
a Copy execution C, the first entry in ω with value p is one of
those output entries of C.

C: For any structure operation execution e initiated in ω and for any

Assign, Update, or Delete execution A, e is in reach R(A) in ω $\Rightarrow$

A is initiated before e.

Proof: By induction on the lengths of the prefixes α of ω. Induction

hypotheses are that A and B are true for any p which appears as the value

of an entry in α, and C is true for any e initiated in α.

Basis: $|\alpha| = 0$. A and B are vacuously true.

(1) e = Ex(d,k) is a structure operation execution $\Rightarrow$ In(I(d)) > 0 $\Rightarrow$

    e is not initiated in α Defs. 5.1-1+4.2-6

Induction step: Assume the induction hypotheses are true for any prefix

of length n, $0 \leq n < |\omega|$, and consider prefix αf of ω of length n+1.

(2) Let p be any pointer which is the value of an entry in αf. If p

    is the value of an entry in α, then A and B hold for p   ind. hyp.

(3) Assume that p is not the value of any entry in α. Then f is the

    first entry in ω with value p   (2)

(4) Let e be the execution of which f is an output entry. Either e is

    in IE, e is a pI execution, or e is a Copy or Select execution

                                         (2)+(3)+Const. 5.1-1

(5) e is initiated in α (4)+Def. 4.2-7

(6) e is a pI execution $\Rightarrow$ $\exists j$: V(Ent(e,j)) = p (3)+(4)+Defs. 4.2-6+5.1-2

(7) $\Rightarrow$ there is an entry in α with value p   (5)+Def. 4.2-6

(8) e is not a pI execution   (6)+(7)+(3)

(9) e is a Select execution which is in a reach in ω $\Rightarrow$ e is in the reach

    of an Update execution U and V(Ent(U,3)) = p

                                 (3)+(4)+Def. 6.1-6+Const. 5.1-4

(10) $\Rightarrow$ U is initiated before e in ω (5)+Def. 4.2-6+ind. hyp. C

(11) ⇒ U is initiated in α, so Ent(U,3) is in α   (5)+Def. 4.2-6

(12) e is not a Select execution which is in a reach in ω   (9)+(11)+(3)

(13) e either is in IE, is a Copy execution, or is a Select execution

      which is in no reach in ω   (4)+(8)+(12)

(14) p is the value in ω of the output entries of a Copy execution C ⇒

      p is not the value in ω of the output entries of an execution

      which either is in IE, is a Copy execution other than C, or is a

      Select execution which is in no reach in ω   Const. 5.1-7

(15) ⇒ e = C   (3)+(4)+(13)

(16) A and B are true for any p which is the value of an entry in αf

                               (1)+(3)+(13)+(4)+(14)+(15)

(17) Let e be any structure operation execution initiated in αf. e is

      initiated in α ⇒ C is true for e   ind. hyp. C

(18) Assume f is the initiating entry of e in ω. Let A be any Assign,

      Update, or Delete execution such that e∈R(A) in ω. Then Ent(e,1)

      is in duration D(A) in ω   Defs. 5.1-6+5.1-8

(19) Either

(19a) Ent(A,1) precedes Ent(e,1) in the same access history in ω, or

(19b) Ent(e,1) is in access history H_p^ω for some p which is the value of

      the output entries in ω of a Copy execution C, and Ent(C,1) is in

      D(A) in ω   (18)+Defs. 5.1-5+5.1-7

(20) (19a) ⇒ A initiates before e in ω   Def. 5.1-4

(21) (19b) ⇒ C∈R(A)   Defs. 5.1-6+5.1-8

(22) ∧ There is an entry in αf with value p   (18)+Defs. 5.1-4+4.2-6

(23) ∧ the first entry with value p in ω is an output entry of C

                               (19b)+(16)

(24) ⇒ there is an output entry of C in αf   (22)

(25) ⇒ C is initiated in α   Def. 4.2-7

(26) ⇒ A is initiated before C in α   (24)+ind. hyp. C

(27) ⇒ A is initiated in α   (25)+Def. 4.2-6

(28) ⇒ A is initiated before e in ω   (18)+Def. 4.2-6

(29) C is true for any e initiated in αf   (17)-(21)+(28)

$\triangle$

Lemma 5.3-9 Let α and β be any two causal computations for the same interpretation (St, I,IE) such that β is SOE-inclusive of α. If β satisfies the Input/Output Type, Structure Output, and Unique Pointer Generation Constraints, then α satisfies these constraints.

Proof:

(1) Let $f = Ent_\alpha(e,j)$ be any entry in α, and let $e = Ex(d,k)$. $I(d)$ is a pI action ⇒ the types of the input entries of e are not constrained   Const. 5.1-1

(2) $I(d)$ is not a pI actor and e has input entries in α ⇒ for all j, $V(Ent_\beta(e,j)) = V(Ent_\alpha(e,j))$ (1)+Def. 5.2-8

(3) The type of $V(Ent_\beta(e,j))$ depends on $I(d)$ and j as in Const. 5.1-1, so the type of $V(Ent_\alpha(e,j))$ depends on $I(d)$ and j as in Const. 5.1-1   (2)

(4) Let the source in T(f) be Src(e',i), where $e' = Ex(d',k')$. Then there is an entry g in β such that V(g) = V(f) and T(g) has the same source; i.e., the value of Src(e',i) is the same in α and β   (1)+Defs. 5.2-8+4.2-6

(5) The type of V(g) depends on $I(d')$ and i as in Const 5.1-1, so the type of V(f) depends on $I(d')$ and i as in the constraint   (4)

(6) α satisfies the Input/Output Type Constraint(1)+(3)+(5)+Const. 5.1-1

(7) For any pointer p, p is the value in β of the output entries of a

Copy execution C $\Rightarrow$ the first entry in β with value p is one of

those output entries of C   Lemma 5.3-8

(8) For any Update or Delete execution U, for j=2,3, if there is an

entry $Ent_\alpha(U,j)$, then there is an entry $Ent_\beta(U,j)$, and they have

the same value   Def. 5.2-8

(9) For any Select, Update, or Delete execution e initiated in α,

$e \in R(U)$ in α iff $e \in R(U)$ in β (7)+Lemma 5.2-6

(10) $e \in R(U)$ in α $\Rightarrow$ $e \in R(U)$ in β $\Rightarrow$ the values of Src(e,1) and Src(e,2) in

β depend on $V(Ent_\beta(U,2))$, and possibly on $V(Ent_\beta(U,3))$, as in

Constraint 5.1-4   (9)+Const. 5.1-4

(11) $\Rightarrow$ the values of Src(e,1) and Src(e,2) in α depend on $V(Ent_\alpha(U,2))$,

and possibly on $V(Ent_\alpha(U,3))$, as in the constraint   (8)+(4)

(12) α satisfies the Structure Output Constraint   (10)+(11)+Const. 5.1-4

(13) Let C be any Copy execution initiated in α, and let p be the value

of C's output entries in α (if any). α does not satisfy the

Unique Pointer Generation Constraint $\Rightarrow$ there is an execution e$\neq$C

whose output entries have value p in α and e either is in IE, is

a Copy execution, or is a Select execution which is in no reach

in α Const. 5.1-7

(14) $\Rightarrow$ C and e have output entries of value p in β (1)+(4)+Def. 4.2-5

(15) $\wedge$ if e is a Select execution, it is not in a reach in β (9)

(16) $\Rightarrow$ β does not satisfy the Constraint   Const. 5.1-7

(17) α satisfies the Unique Pointer Generation Constraint   (13)+(16)

$\triangle$

__Lemma 5.3-10__ Given an interpretation Int = (St, l, IE), let ρ be the equal

pointer relation defined from Int. Let a_1, a_2, ω_1, and ω_2 be four causal

computations for Int such that either for i=1,2, a_i is a prefix of a

permutation of ω_i, or for i=1,2, ω_i is SOE-inclusive of a_i. If

(1) for i=1,2, for every structure operation execution e initiated before

the last entry in a_i, and every Assign, Update, or Delete execution

A, e is in the reach R(A) in a_i iff e is in R(A) in ω_i,

then for any two pointers p_1 and p_2, $(p_1,a_1)\rho(p_2,a_2) \Rightarrow (p_1,\omega_1)\rho(p_2,\omega_2)$.

__Proof:__

(2) a_i is a prefix of a permutation of ω_i $\Rightarrow$ every entry in a_i is in ω_i

(3) For every execution e which is in IE or is a structure operation

execution, for any integer j and any value v, there is an entry

$f \in a_i$ such that T(f) has source Src(e,j) (destination Dst(e,j)) and

$V(f) = v \Rightarrow$ there is an entry g in ω_i such that T(g) has source

Src(e,j) (destination Dst(e,j)) and V(g) = v   (2)+Def. 5.2-8

(4) $(p_1,a_1)\rho(p_2,a_2)$ iff

(4a) there is a source s = Src(e,i), for some e∈IE and some i, such that

p_1 (p_2) is the value of s in a_1 (a_2), or

(4b) there are Select execution S_1 and S_2 such that

p_i is the value of Src(S_i,1) in a_i, i=1,2,

S_i is not in any reach in a_i, i=1,2,

$V(Ent_{a_1}(S_1,2)) = V(Ent_{a_2}(S_2,2))$, and

$(V(Ent_{a_1}(S_1,1)),a_1)\rho(V(Ent_{a_2}(S_2,1)),a_2)$, or

(4c) there is a pointer $q \neq p_1$ such that $DD_{a_1}(q,p_1)$ and $(q,a_1)\rho(p_2,a_2)$

Def. 5.1-10

The proof of the Lemma is by induction on n, the number of recursive

applications of the above three rules necessary to derive $(p_1,\alpha_1)\rho(p_2,\alpha_2)$.

(5) (4a) is true of p_1 and $p_2 \Rightarrow$ there is a one-step derivation, so n = 1

(6) The last step in the shortest derivation is an application of (4b)

or (4c) $\Rightarrow$ there is a pair of pointers q_1 and q_2, not the same as

p_1 and p_2, such that it has been derived that $(q_1,\alpha_1)\rho(q_2,q_2)$

(7) $\Rightarrow$ there is at least one additional step in the shortest derivation,

so n > 1

Basis: n = 1

(8) (4a) is true of p_1 and p_2 (4)+(6)+(7)

(9) All entries in α_1 (α_2) whose transfers have source s have value

p_1 (p_2)   (4a)+Def. 4.2-6

(10) All entries in ω_1 (ω_2) whose transfers have source s have value

p_1 (p_2)   (9)+(4a)+(3)

(11) $(p_1,\omega_1)\rho(p_2,\omega_2)$ (10)+Defs. 4.2-6+5.1-10

Induction step: Assume that the Lemma is true for any p_1 and p_2 if the

shortest derivation of $(p_1,\alpha_1)\rho(p_2,\alpha_2)$ has n steps, n>0, and consider

(12) p_1 and p_2 for which the shortest derivation has n+1 steps

(13) Either (4b) or (4c) is applied as the last step in this   (4)+(5)

(14) (4b) $\Rightarrow$ S_i has output entries in α_i Def. 4.2-6

(15) $\Rightarrow$ S_i is initiated before the last entry in α_i Def. 4.2-7

(16) $\wedge$ p_i is the value of $Src(S_i,1)$ in ω_i, for i=1,2   (3)+Def. 4.2-6

(17) $\Rightarrow$ S_i is not in any reach in ω_i (15)+(1)

(18) $\wedge$ $V(Ent_{\omega_1}(S_1,2)) = V(Ent_{\omega_2}(S_2,2))$ (3)+Def. 4.2-6

(19)   Letting $q_i = V(Ent_{\alpha_i}(S_i,1)) = V(Ent_{\omega_i}(S_i,1))$, i=1,2, (4b) $\Rightarrow$

$(q_1, a_1) \rho(q_2, a_2) \Rightarrow$ a shortest derivation of $(p_1, a_1) \rho(p_2, a_2)$ consists

of a shortest derivation of $(q_1, a_1) \rho(q_2, a_2)$ followed by the

application of (4b)   (3)+(13)+Def. 4.2-6

(20) $\Rightarrow (q_1, \omega_1) \rho(q_2 \cdot \omega_2)$ (12)+ind. hyp.

(21) (4b) $\Rightarrow (p_1, \omega_1) \rho(p_2, \omega_2)$ (16)-(20)+Def. 5.1-10

(22) For any pointer $q \neq p_1$, $DD_{a_1}(q, p_1) \Rightarrow$ there is a sequence of Copy

executions $C_1, \ldots, C_m$ such that $V(Ent_{a_1}(C_1, 1)) = q$, p_1 is the value

of the output entries of C_m, and if $m > 1$, then for $j = 2, \ldots, m$,

$V(Ent_{a_1}(C_j, 1))$ is the value of the output entries of C_{j-1} Def. 5.1-9

(23) $\Rightarrow$ since each of $C_1, \ldots, C_m$ has input and output entries of the same

value in ω_1, $DD_{\omega_1}(q, p_1)$ (3)+Def. 5.1-9

(24) (4c) $\Rightarrow \exists q \neq p_1$: $DD_{\omega_1}(q, p_1)$ and $(q, a_1) \rho(p_2, a_2)$ (22)+(23)

(25) $\Rightarrow$ a shortest derivation of $(p_1, a_1) \rho(p_2, a_2)$ consists of a shortest

derivation of $(q, a_1) \rho(p_2, a_2)$ followed by an application of (4c) (13)

(26) $\Rightarrow (q, \omega_1) \rho(p_2, \omega_2)$ (12)+ind. hyp.

(27) (4c) $\Rightarrow (p_1, \omega_1) \rho(p_2, \omega_2)$ (24)+(26)+Def. 5.1-10

(28) (12) $\Rightarrow (p_1, \omega_1) \rho(p_2, \omega_2)$ (13)+(21)+(27)

$\triangle$

5.3.4 Conclusion

This subsection concludes the proof that all computations in a job satisfy the last five constraints. The third and final step in that proof has been explained at the start of Section 5.3; it is here repeated precisely as:

Lemma 5.3-11 Let S_1 and S_2 be any two equal initial standard states for the same L_{BS} program P and let Ω_1 and Ω_2 be any two halted firing sequences

starting in S_1 and S_2 respectively. Let $\omega_1 = \eta(S_1, \Omega_1)$ and $\omega_2 = \eta(S_2, \Omega_2)$, and assume that these are computations for Int(P). Let α_1 and α_2 be any two causal computations for Int(P) and let ρ be the equal pointer relation defined from Int(P). If, given Int(P),

(1) for i=1,2, for any structure operation execution e, e is initiated in $\alpha_i \rightarrow$ e is initiated in ω_i, for every integer j, if there is an entry $Ent_{\alpha_i}(e,j)$ in α_i, then there is an $Ent_{\omega_i}(e,j)$ in ω_i with the same value, and if there is an entry in α_i whose transfer has source Src(e,j), then there is an entry in ω_i with the same value whose transfer has source Src(e,j),

(2) for i=1,2, for every structure operation execution e initiated in α_i and any Assign, Update, or Delete execution A, e$\in$R(A) in α_i iff e$\in$R(A) in ω_i, and

(3) for any pointers p_1 and p_2, $(p_1, \alpha_1)\rho(p_2, \alpha_2) \Rightarrow (p_1, \omega_1)\rho(p_2, \omega_2)$,

then α_i satisfies the Atomic Output, Structure Output, and Unique Pointer Generation Constraints, and the pair consisting of α_1 and α_2 satisfies the Initial Structure and the First/Next Output Constraints.

Proof: (Since the earlier explanation is conceptually complete, the details of the proof have been relegated to Appendix D.) $\triangle$

All of the elements presented in this section are now brought together, to verify that:

Theorem 5.3-3 EE(L_{BS},S) is a Structure-as-Storage model.

Proof:

(1) $EE(L_{BS},S) = (V, L, A, In, E)$ is an entry-execution model   Thm. 4.3-1

(2) There is a distinct subset V_p of V containing pointers

                                                          Defs. 2.2-1+4.3-1

(3) The action domain A contains the following eight actions, and In

      assigns to each the indicated input arity: Fetch(1), First(1),

      Next(2), Select(2), Copy(1), Assign(2) Update(2), and Delete(2)

                                               Defs. 2.2-3+2.2-5+4.3-1

(4) Let (Int,J) be any expansion in E. Then there is an L_{BS} program P

      such that this is an expansion of P   Def. 4.3-1

(5) Let J be any job in J. Then J is a job for Int   (1)+(4)+Def. 4.2-3

(6) Int = Int(P) and there is an equivalence class E of initial

      standard states for P such that $J = J_E$ (4)+Def. 4.3-2

(7) Let S_1 and S_2 be any two states in E, and let Ω_1 and Ω_2 be any two

      halted firing sequences starting in S_1 and S_2. For i=1,2, let β_i

      be any computation in J_{S_i,Ω_i}. Then β_i is a causal permutation of

      $\omega_i = \eta(S_i,\Omega_i)$ Def. 4.3-5

(8) ω_i is also in J_{S_i,Ω_i} and $\Phi(\omega_i)$ is the reduction of Ω_i Lemma 4.3-3

(9) ω_i and β_i are both in J_E, hence both are computations for Int=Int(P)

                                       (7)+(8)+(5)+(6)+Defs. 4.3-3+4.2-3

(10) ω_i is SOE-inclusive of β_i (7)+Lemma 5.3-7

(11) ω_i is causal   (7)+Lemma 4.3-2

(12) ω_i satisfies the Input/Output Type Constraint, given Int(P)

                                       (7)+Lemma 5.3-1

(13) ω_i satisfies the Structure Output Constraint, given Int(P)

                                       (7)+Lemma 5.3-3

(14) ω_i satisfies the Unique Pointer Generation Constraint, given Int(P)

$\hspace{8cm}$ (7)+Lemma 5.3-6

(15) For any pointer p, p is the value of the output entries in ω_i of a

Copy execution C $\Rightarrow$ the first entry in ω_i with value p is one of

those output entries of C $\hspace{2.5cm}$ (11)+(9)+(12)-(14)+Lemma 5.3-8

(16) For any structure operation execution e initiated in β_i, and any

Assign, Update, or Delete execution A, e$\in$R(A) in β_i $\Leftrightarrow$ e$\in$R(A) in ω_i

$\hspace{4cm}$ (7)+(11)+(9)+(10)+(15)+Lemma 5.2-6

(17) β_i satisfies the Input/Output Type, Structure Output, and Unique

Pointer Generation Constraints given Int(P)

$\hspace{5cm}$ (7)+(9)+(10)-(14)+Lemma 5.3-9

(18) For any pointer p, p is the value of the output entries in β_i of the

Copy execution C $\Rightarrow$ the first entry in β_i with value p is one of

those output entries of C $\hspace{3cm}$ (7)+(9)+(17)+Lemma 5.3-8

(19) Let α_i be any prefix of β_i. Let γf be any prefix of α_i and let e be

the execution of which f is an output entry. Then γf is a prefix

of β_i, so e is initiated in γ $\hspace{3cm}$ (7)+Def. 4.2-7

(20) α_i is causal $\hspace{6cm}$ (19)+Def. 4.2-7

(21) α_i is in J_E, and so is a computation for Int(P)

$\hspace{4cm}$ (19)+(7)+(5)+Defs. 4.3-3+4.2-3

(22) For any structure operation execution e = Ex(d,k) initiated in α_i,

and any Assign, Update, or Delete execution A, e$\in$R(A) in α_i iff

e$\in$R(A) in β_i $\hspace{2cm}$ (20)+(7)+(21)+(9)+(19)+(18)+Lemma 5.2-5

(23) $\wedge$ there are In(I(d)) input entries to e in α_i, hence in β_i, so e is

initiated in β_i $\hspace{5cm}$ (19)+Def. 4.2-6

(24) $\Rightarrow$ e$\in$R(A) in α_i iff e$\in$R(A) in ω_i (16)

(25) Let f be any entry in α_i. Then f is in β_i (22)

(26) Let T(f) be (Src(Ex(d,k),i), Dst(Ex(d',k'),j)). Constraint 5.1-1

dictates, one or two times, what the type of V(f) should be: once

based on l(d) and i, and again based on l(d') and j   Const. 5.1-1

(27) Both of the types so dictated match the type of V(f) (25)+(17)

(28) α_i satisfies the Input/Output Type Constraint (25)+(21)+Const. 5.1-1

(29) J satisfies the Pointer Transparency Constraint (4)+(5)+Lemma 5.3-2

(30) Let e be any structure operation execution. If e is initiated in α_i

there are In(l(d)) input entries to e in α_i, so the same entries

are in β_i and ω_i, so e is initiated in ω_i. For every integer j,

if there is an entry Ent$_{\alpha_i}$(e,j) in α_i, then there is an entry

Ent$_{\omega_i}$(e,j) in ω_i with the same value. If there is an entry in α_i

whose transfer has source Src(e,j), there is one in ω_i with the

same value whose transfer has the same source (25)+(7)+Def. 4.2-6

(31) Let ρ be the equal pointer relation defined from Int. For any two

pointers p_1 and p_2, $(p_1,\alpha_1)\rho(p_2,\alpha_2)$ $\Rightarrow$ $(p_1,\beta_1)\rho(p_2,\beta_2)$

(20)+(7)+(21)+(9)+(19)+(22)+Lemma 5.3-10

(32) $\Rightarrow$ $(p_1,\omega_1)\rho(p_2,\omega_2)$ (7)+(11)+(9)+(10)+(16)+Lemma 5.3-10

(33) Given Int(P), α_i satisfies the Atomic Output, Structure Output, and

Unique Pointer Generation Constraints, and the pair consisting of

α_1 and α_2 satisfies the Initial Structure and First/Next Output

Constraints (7)+(9)+(20)+(21)+(30)+(22)+(24)+(31)+(32)+Lemma 5.3-11

(34) EE(L_{BS},S) is an S-S model

(1)-(6)+(7)+(19)+(28)+(33)+(29)+Defs. 4.3-3+5.1-1

Q.E.D.

Chapter 6

A Generalized Determinacy Proof

A novel method for guaranteeing determinacy of L_{BS} programs has been presented in Chapter 3. Its key feature is the withholding of read pointers (p,R) output by a Select firing so long as there are write pointers (p,W) on arcs of the program. It was argued that this scheme guarantees freedom from conflict; the purpose of this chapter and the following is to show that such freedom implies determinacy, and, in turn, functionality.

The entry-execution model was introduced in Chapter 4 as being particularly well-suited to the statement and proof of assertions about specific operations. Chapter 5 has illustrated its use in concisely describing the operations characterizing a Structure-as-Storage language. The current chapter presents, in entry-execution terms, a set of seven Determinacy Axioms for an S-S model, and proves that they are sufficient to guarantee determinacy. Chapter 7 then shows that the model $EE(L_D,M)$ satisfies these axioms, hence is determinate, and that this implies the functionality of L_D programs run on the modified interpreter.

This chapter commences with the formal definition of determinacy in an entry-execution model (Section 6.1). The Determinacy Axioms are presented next (Section 6.2). The final two sections contain the proof that the Axioms are sufficient for determinacy in any S-S model.

6.1 The Definition

A rough definition of determinacy in the standard model of data flow has already been given, in Section 3.1.2 (q.v.). With the elegant vocabulary of the entry-execution model, this is easily translated into a more concise statement.

Determinacy is a property of individual programs in a data-flow language. To each such program P there corresponds an expansion (Int, J) in an entry-execution model of that language. Therefore, in that model, determinacy is a property of expansions. Program P is determinate iff, for each equivalence class E of initial states for P, any two firing sequences starting in any states in E satisfy the five Determinacy Assertions. Each such equivalence class E corresponds to a single job J_E in J. A job is just a set of computations, which are the entry-execution analogs of firing sequences. Therefore, an expansion is determinate iff, for each job $J \in J$, any two computations in J satisfy corresponding conditions.

The first and fourth of the Determinacy Assertions together state: For any two actors d_1 and d_2 and any integers i, j, k, and m, there is a kth firing of d_1 and an mth firing of d_2, and a value is transferred from the number-i output of the latter to the number-j input of the former, in firing sequence Ω_1 iff the same is true of firing sequence Ω_2. The corresponding condition on two computations ω_1 and ω_2 is: There is an entry with transfer $(Src(Ex(d_2,m),i), Dst(Ex(d_1,k),j))$ in ω_1 iff there is an entry with that transfer in ω_2.

The second and third Determinacy Assertions concern the value of the number-j input to the kth firing of an actor d. These translate directly into statements about the values of the entries in ω_1 and ω_2 whose transfers have destination $Dst(Ex(d,k),j)$ (of which there is at most one per computation). To wit: There is a one-to-one map F over pointers such that, for any entries f in ω_1 and g in ω_2 with $T(g) = T(f)$,

a. $V(f)$ is not a pointer iff $V(g)$ is not a pointer,

b. if those values are not pointers, then they are the same, and

c. if those values are pointers, then $F(V(f))$ is defined and equal to $V(g)$.

The final Determinacy Assertion is of the equality of the reaches in Ω_1 and Ω_2 of each Assign, Update, or Delete firing. The concept of reach has already been refined and translated into entry-execution terms in Chapter 5. The requirement for equal reaches is combined with those translated above to form:

Definition 6.1-1 An expansion (Int,J) is determinate iff for each job $J \in J$, any two computations in J are equivalent under some one-to-one pointer correspondence.

A pointer correspondence F is any map over pointers

$$F: V_p \to V_p$$

Two computations ω_1 and ω_2 are equivalent under pointer correspondence F iff the following are all true:

1. The sets of transfers of the entries in ω_1 and ω_2 are identical.

2. For any two entries $f \in \omega_1$ and $g \in \omega_2$ such that $T(g) = T(f)$,

    a. $V(f)$ is not a pointer iff $V(g)$ is not a pointer,

    b. if those values are not pointers, then they are the same, and

    c. if those values are pointers, then $F(V(f))$ is defined and equal to $V(g)$.

3. For any Assign, Update, or Delete execution A, the reach $R(A)$ in ω_1 equals the reach $R(A)$ in ω_2.

$\triangle$

6.2 The Axioms

This section presents the seven Determinacy Axioms for an S-S model. These include the six Determinate Schema Axioms, plus freedom from conflict between structure operations. The Determinate Schema Axioms are sufficient to guarantee determinacy of programs in languages without structure operations; consequently, they are well-understood and have been made to hold by design for most existing parallel-programming languages. These are succinctly presented in terms of a state-transition model of computation in Denning [9]; here they are translated into entry-execution terms.

The first two axioms are causality and the Prefix Property. These are implicit in the state-transition paradigm, and they also hold for the one entry-execution model which has been constructed. They are not inherent in the entry-execution view, however, and so should be explicit.

__Axiom 6.2-1__ (Causality) For any expansion (Int, J), every computation in every job in J is causal with respect to Int.

$\triangle$

<u>Axiom 6.2-2</u> (Prefix Property) For any expansion (Int,J), every job in possesses the Prefix Property.

$\triangle$

The third axiom states simply that any action except a structure operation is deterministic; i.e., in all computations in which it has the same set of input values, it produces the same set of output values.

<u>Definition 6.2-1</u> Given any expansion (Int,J) where Int = (St, I,IE), any action a is <u>deterministic</u> iff the following is true for any two (not necessarily distinct) computations ω_1 and ω_2 in any two jobs in J: For i=1,2, let e_i = Ex(d_i,k_i) be any execution not in IE such that $I(d_i)$ = a. Then

> for all j, there is an entry Ent(e_1,j) in ω_1 iff there is an entry
>> Ent(e_2,j) in ω_2, and if so, those entries' value are equal
> $\Rightarrow$ for all i, the value of Src(e_1,i) in ω_1 (if any) equals the value
>> of Src(e_2,i) in ω_2 (if any).

$\triangle$

<u>Axiom 6.2-3</u> (Determinism) For any expansion, all actions except the eight structure operations (Fetch, First, Next, Select, Assign, Update, Delete, and Copy) are deterministic.

$\triangle$

The fourth Determinacy Axiom completes the characterization, begun in Chapter 4, of a job as the set of computations by a single program on a particular set of inputs. Reviewing the development to this point: The set of initial interpreter states which represent that program and set of inputs constitutes an equivalence class E. An individual program input X is represented in an initial state by the value of a token on a particular

program input arc b of P. In a program with no structures, X is the same in two initial states iff the tokens on b have identical non-pointer values. However, the same structure input can be represented by different pointers in different states in E (if the components of the heaps to which they point are equal).

In the entry-execution model of data flow, P corresponds to an expansion (Int, J), where $Int = (St, I, IE)$, and E gives rise to a job J_E in J. The values residing on program input arcs of P in initial states in E are represented as the values of output entries of executions in IE in the computations in J_E. Because of the disparity between pointer- and non-pointer-valued inputs, the definition of job places no restrictions on the values of output entries of executions in IE.

The pointer-valued output entries of an execution in IE may have arbitrarily-different values p_1 and p_2 in different computations ω_1 and ω_2 in J_E. But those pointers are related; in particular, $(p_1, \omega_1) \rho (p_2, \omega_2)$. The significance of this relationship is evident in the constraints on the output entries in ω_1 and ω_2 of executions having p_1 and p_2 as inputs (Constraints 5.1-4 and 5.1-5).

No constraints on the output entries of executions in IE are necessary in a general entry-execution model. Constraints 5.1-4 and 5.1-5 on pointer-valued entries are necessary in any S-S model. And the following constraint on the heretofore-unspecified non-pointer-valued entries is necessary in any determinate entry-execution model.

<u>Axiom 6.2-4</u> For any expansion (Int,J) where Int = (St,I,IE), for any e∈IE, any integer i, and any two computations ω_1 and ω_2 in a job in J, the value of Src(e,i) in ω_1 is not a pointer iff the value of Src(e,i) in ω_2 is not a pointer, and if those values are both not pointers, then they are equal.

$$\triangle$$

The next two axioms concern aspects of the control structure and local-memory structure of a program. Both of these make use of the concept of <u>eligible</u> <u>transfers</u>; analogous to enabled operators in a state, the eligible transfers at the end of a computation α are just those transfers which can immediately follow α in some longer computation:

<u>Definition 6.2-2</u> Given a job J and any computation α in J, the set $ET_J(\alpha)$ of <u>eligible</u> <u>transfers</u> (<u>at</u> <u>the</u> <u>end</u> <u>of</u> α) is defined by

$$ET_J(\alpha) = \{t \mid \exists f: T(f) = t \text{ and } \alpha f \in J\}$$

$$\triangle$$

The first of the two axioms combines a pair of Denning's: persistence and non-interference. In a state-transition model of a language like data flow, persistence means that once an operator is enabled to fire, it cannot be disabled by subsequent firings of other operators; i.e., it remains enabled until it fires, and it must fire before the firing sequence can halt. Non-interference concerns the sources of the values transferred to the number-i input of the jth firing of actor d_1 in different firing sequences. If in one firing sequence, that value is transferred from the output of the kth firing of actor d_2, then in any other firing sequence in which there is a jth firing of d_1 and a value is transferred to its number-i input, that transfer is from the output of the kth firing of d_2.
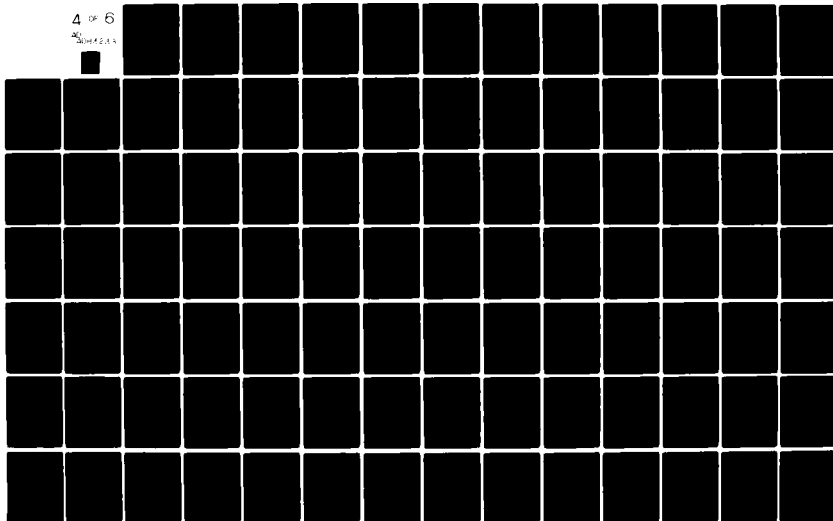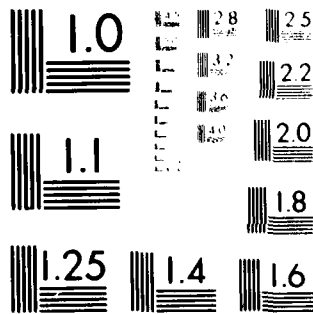
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

In an entry-execution model, the existence of a jth firing of d_1 is represented as the existence of a complete set of input entries to the execution $Ex(d_1,j)$. Therefore, both persistence of firings and non-interference between transfers can be combined: Once a firing is enabled, a set of input entries to the corresponding execution becomes eligible. Persistence of firings implies that some set of input entries remains eligible, regardless of any subsequent entries, until it occurs. Furthermore, non-interference means that the sources of the transfers of those entries are the same, regardless of when the entries occur. Thus, once a transfer is eligible, that same transfer — same destination, same source — remains eligible until it occurs. I.e., eligibility of transfers is persistent:

Axiom 6.2-5 (Persistence) For any expansion (Int,J), for any job $J \in J$ and any computation αg in J, for any transfer $t \neq T(g)$, $t \in ET_J(\alpha) \rightarrow t \in ET_J(\alpha g)$.

$\triangle$

This axiom is inductively extended in the following lemma.

Lemma 6.2-1 For any persistent expansion (Int,J), let ω be any computation in any $J \in J$ and let t be any transfer in $ET_J(\omega)$. Then for any γ such that $\omega\gamma$ is in J:

$$\nexists f \in \gamma: T(f) = t \rightarrow t \in ET_J(\omega\gamma)$$

Proof: By induction on the length of γ.

Basis: $|\gamma| = 0$. Then $ET_J(\omega\gamma) = ET_J(\omega)$, so $t \in ET_J(\omega\gamma)$.

Induction step: Assume the Lemma is true for any γ of length $n \geq 0$, and consider

(1) $\omega\gamma = \omega\delta g$, where $|\gamma| = n+1$

(2) $\not\exists f \in \gamma: T(f) = t \Rightarrow \not\exists f \in \delta: T(f) = t \land T(g) \neq t$ (1)

(3) $\Rightarrow t \in ET_J(\omega\delta)$ ind. hyp.

(4) $\Rightarrow t \in ET_J(\omega\delta g)$ (2)+Ax. 6.2-5

$$\triangle$$

The second axiom concerned with control is commutativity. Denning states that if two adjacent firings in a sequence can be swapped, then doing so should not change the resultant control state. I.e., if $\Omega\varphi_1\varphi_2$ and $\Omega\varphi_2\varphi_1$ are both firing sequences starting in state S, then at least the control portions of the states $S \cdot \Omega\varphi_1\varphi_2$ and $S \cdot \Omega\varphi_2\varphi_1$ are equal. Beyond this explicit axiom, there is implicit in the state-transition paradigm the assumption that firing the same actor in either of two equal control states results again in equal control states. I.e., if $S \cdot \Omega\varphi_1\varphi_2$ and $S \cdot \Omega\varphi_2\varphi_1$ have equal control portions, then for any θ such that both $\Omega\varphi_1\varphi_2\theta$ and $\Omega\varphi_2\varphi_1\theta$ are firing sequences starting in S, $S \cdot \Omega\varphi_1\varphi_2\theta$ and $S \cdot \Omega\varphi_2\varphi_1\theta$ have equal control portions. Both of these assertions — Denning's explicit axiom and the tacit assumption about state transitions — must be made explicit in the entry-execution model; the following axiom combines them into a single simple statement:

Axiom 6.2-6 (Commutativity) For any expansion (Int, J), for any job $J \in J$, and for any computation $\alpha g f \delta$ in J such that $\alpha f g \delta$ is also in J,

$$ET_J(\alpha f g \delta) = ET_J(\alpha g f \delta)$$

$$\triangle$$

The seventh and final Determinacy Axiom is the freedom-from-conflict axiom. This principle has already been explained (at the end of Section 3.1), and is here translated into entry-execution terms with the aid of the following observation: Let φ_1 and φ_2 be two firings in a firing sequence Ω starting in state S such that, for $i=1,2$, φ_i is the $k_i{}^{th}$ firing of actor d_i. Let e_1 and e_2 be the executions $Ex(d_1,k_1)$ and $Ex(d_2,k_2)$ respectively. If φ_1 and φ_2 potentially interfere, then in $\eta(S,\Omega)$, $Ent(e_1,1)$ and $Ent(e_2,1)$ are in the same access history, and one of e_1 and e_2 is in the reach of the other.

<u>Axiom 6.2-7</u> (Freedom from conflict) For any expansion (Int,J), for any job $J \in J$, there is no computation αgf in J satisfying all the following:

1. f and g initiate *distinct* executions e_1 and e_2 respectively in αgf,

2. $Ent(e_1,1)$ and $Ent(e_2,1)$ are in the same access history in αgf,

3. e_1 is in the reach $R(e_2)$ in αgf, and

4. there is a computation $\alpha \bar{f}\bar{g}$ in J with $T(\bar{f}) = T(f)$ and $T(\bar{g}) = T(g)$.

$\triangle$

6.3 The Basic Requirements for Equivalence of Computations

This section presents an important general result concerning the equivalence of two computations ω_1 and ω_2 in a job which satisfies the first four of the Determinacy Axioms: The first part of the definition of equivalence — identical sets of transfers — plus one additional simple condition together imply the remaining three components of equivalence — equal non-pointer values, corresponding pointer values, and equal reaches. The reason for presenting this here as a separate significant

result is that it is anticipated that future research will explore issues
other than strict determinacy which concern equivalence; it is hoped that
these endeavors will be aided by the pre-existence of such a general proof.

The second of the two conditions sufficient for equivalence is set
forth in the following:

Definition 6.3-1 Given a job J, let α and β be any two computations in J.
Then β preserves the order of dependent accesses in α iff the following
is true of every structure operation execution e: Let A
be any Assign, Update, or Delete execution. If $Ent_\alpha(A,1)$ and $Ent_\alpha(e,1)$
are in the same access history in α and e is in the reach R(A) in α, then
A initiates before e in β. If $Ent_\beta(A,1)$ and $Ent_\beta(e,1)$ are in the same
access history in β and e is in R(A) in β, then A initiates before e in α.

$\triangle$

Relating this to the firing sequences modeled by α and β, Ent(A,1)
and Ent(e,1) are in the same access history iff the firings which they
represent input the same pointer, i.e., access the same node. If e is in
the reach of A, then the state change effected by the firing represented
by e depends upon the inputs to the firing represented by A. Thus the
property just defined is analogous to the following relationship between
firing sequences: If in one firing sequence, the result of a firing which
accesses node n depends upon another firing which also accesses node n,
then those firings occur in the same order in the other firing sequence.

The proof that this together with equal transfer sets imply the three
remaining components of equivalence proceeds by induction on the lengths

of the prefixes of one of the computations. Because it is very long, this proof is broken into three lemmas, one for each component. This requires that each component be given a name, and furthermore, that a modified version of each be made available to relate one computation to a prefix of the other. This is done is the following series of definitions.

Definition 6.3-2 A computation β is _transfer-inclusive_ of another computation α iff for each entry f in α, there is an entry g in β with the same transfer. Two computations are _transfer-congruent_ iff each is transfer-inclusive of the other.

$\triangle$

Definition 6.3-3 Given two computations α and β such that β is transfer-inclusive of α, β is _NPE-inclusive_ of α iff the following is true of every entry f in α: Let g be the entry in β with $T(g) = T(f)$. Then $V(f)$ is not a pointer iff $V(g)$ is not a pointer, and if those values are not pointers, then they are the same.

$\triangle$

Definition 6.3-4 Given two computations α and β such that β is transfer-inclusive of α, and a pointer correspondence F, β is _PE-inclusive of α under F_ iff the following is true for every entry f in α: Let g be the entry in β with $T(g) = T(f)$. Then $V(f)$ is a pointer iff $V(g)$ is a pointer, and if they are pointers, then $F(V(f))$ is defined and equal to $V(g)$.

$\triangle$

Definition 6.3-5 Computation β is _reach-inclusive_ of computation α iff, for each structure operation execution e initiated in α, and each Assign

Update, or Delete execution A, e is in reach R(A) in α iff e is in R(A) in β.

$\triangle$

Definition 6.3-6 Computation β is <u>inclusive</u> <u>of</u> computation α under pointer correspondence F iff it is transfer-inclusive, NPE-inclusive, PE-inclusive under F, and reach-inclusive of α.

$\triangle$

Finally, the proof given here not only establishes the equivalence of ω_1 and ω_2, but also specifies the pointer correspondence under which they are equivalent. This is the natural pointer correspondence F_{ω_1,ω_2}, defined below:

Definition 6.3-7 Given two computations α and β for the same interpretation such that β is transfer-inclusive of α, the <u>natural</u> <u>pointer</u> <u>correspondence</u> <u>for</u> α <u>and</u> β, $F_{\alpha,\beta}$, is given by:

If p is the value of the output entries of a Copy execution C in α, then $F_{\alpha,\beta}(p)$ is the value of the output entries of C in β,

else if there is a pointer p' such that p' is not the value of the output entries of a Copy execution in β and $(p,\alpha)\rho(p',\beta)$, then $F_{\alpha,\beta}(p) = p'$,

else $F_{\alpha,\beta}(p)$ is undefined.

$\triangle$

There are two simple preliminary results which are needed for this and succeeding proofs. The first has already been established for the particular model $EE(L_{BS},S)$; for generality, it is here shown to be true for every S-S model.

<u>Corollary 6.3-1</u> Let α and β be any causal computations for the same interpretation such that α is a prefix of β, and let e be any structure operation execution initiated in α. Then for any Assign, Update, or Delete execution A, e is in the reach R(A) in β iff e is in R(A) in α.

<u>Proof</u>: For any pointer p, p is the value of the output entries in β of a Copy execution C $\Rightarrow$ the first entry in β with value p is one of those output entries of C [Lemma 5.3-8]. The Corollary then follows directly from Lemma 5.2-6.

$\triangle$

The second preliminary result first states that any natural pointer correspondence is one-to-one. It then goes on to show the following fundamental relationship between two equivalent computations α and β: For any pointer p, $(p,\alpha)\rho(F_{\alpha,\beta}(p),\beta)$. The importance of this will become apparent in the proof in Chapter 7 that only a functional L_{BS} program can give rise to a determinate expansion in $EE(L_D,M)$.

<u>Theorem 6.3-1</u> Let α and β be any two causal computations for the same interpretation (St,I,IE) such that β is transfer-inclusive of α. Then $F = F_{\alpha,\beta}$ is one-to-one over the set of pointers over which it is defined. Furthermore, if β is PE-inclusive of α under F, then for each pointer p which is the value of an entry in α, $(p,\alpha)\rho(F(p),\beta)$.

<u>Proof</u>:

(1) Let p be any pointer for which F(p) is defined, and let p' = F(p).
    Then either p (p') is the value in α (β) of the output entries of
    a Copy execution C, or $(p,\alpha)\rho(p',\beta)$ Def. 6.3-7
(2) If p is the value in α of the output entries of a Copy execution C,

then there is a unique pointer p' which is the value in β of the

output entries of C $\qquad$ Defs. 6.3-2+4.2-6

(3) Otherwise, there is a unique p' which is not the value of the output

entries of a Copy execution in β such that $(p,\alpha)\rho(p',\beta)$ Const. 5.1-5

(4) There is a unique p' such that $p' = F(p)$ $\qquad$ (1)+(2)+(3)

(5) Let p_1 and p_2 be any two pointers for which F(p) is defined. One of

the pointers, say p_1, is the value in α of the output entries of a

Copy execution and p_2 is not the value of the output entries in α

of a Copy execution $\Rightarrow F(p_1)$ is the value of the output entries of

a Copy execution in β, $F(p_2)$ is not, and $(p_2,\alpha)\rho(F(p_2),\beta)$ Def. 6.3-7

(6) $\Rightarrow F(p_2)$ is the value in β of the output entries of an execution

which either is in IE or is a Select execution which is in no

reach in β $\qquad$ Def. 5.1-10

(7) $\Rightarrow F(p_1) \neq F(p_2)$ $\qquad$ (5)+Const. 5.1-7

(8) p_1 and p_2 are the values in α of the output entries of Copy

executions C_1 and C_2 respectively $\Rightarrow$ for i=1,2, $F(p_i)$ is the value

in β of the output entries of C_i $\qquad$ Def. 6.3-7

(9) $\Rightarrow [C_1 \neq C_2 \Rightarrow F(p_1) \neq F(p_2)]$ $\qquad$ Const. 5.1-7

(10) $\Rightarrow [F(p_1) = F(p_2) \Rightarrow C_1 = C_2 \Rightarrow p_1 = p_2]$ $\qquad$ Def. 4.2-6

(11) Neither p_1 nor p_2 is the value in α of the output entries of a Copy

execution $\Rightarrow$ neither $F(p_1)$ nor $F(p_2)$ is the value in β of the

output entries of a Copy execution, $(p_1,\alpha)\rho(F(p_1),\beta)$, and

$(p_2,\alpha)\rho(F(p_2),\beta)$ $\qquad$ Def. 6.3-7

(12) $\Rightarrow [F(p_1) = F(p_2) \Rightarrow p_1 = p_2]$ $\qquad$ Const. 5.1-5

(13) $F(p_1) = F(p_2) \Rightarrow p_1 = p_2$ $\qquad$ (5)+(7)+(8)+(10)+(11)+(12)

(14) $F = F_{\alpha,\beta}$ is one-to-one over the set of pointers over which it is

     defined   (5)+(13)

Now prove the second part of the theorem by contradiction. Assume

(15) β is PE-inclusive of α under F, but there is a pointer p which is

     the value of an entry in α and it is not true that $(p,\alpha)\rho(F(p),\beta)$

(16) There is a prefix γf of α such that, for every pointer q which is

     the value of an entry in γ, $(q,\alpha)\rho(F(q),\beta)$, but for $p = V(f)$, it

     is not true that $(p,\alpha)\rho(F(p),\beta)$ (15)

(17) F(p) is defined, and there is an entry in β which is an output

     entry of the same execution as f and has value F(p)   (15)+Def. 6.3-4

(18) p is not the value of the output entries in α of a Copy execution

     $\Rightarrow (p,\alpha)\rho(F(p),\beta)$ (17)+(1)

(19) p is the value of the output entries in α of a Copy execution C

                                                       (18)+(16)

(20) f is the first entry in α with value p   (16)

(21) f is an output entry of C   (19)+(20)+Lemma 5.3-8

(22) $\text{Ent}_\alpha(C,1)$ is in γ (21)+Def. 4.2-7

(23) Let q be $V(\text{Ent}_\alpha(C,1))$. Then $V(\text{Ent}_\beta(C,1))$ is F(q)   (15)+Def. 6.3-4

(24) $(q,\alpha)\rho(F(q),\beta)$ (23)+(22)+(16)

(25) $q \neq p$ and since F is one-to-one, $F(q) \neq F(p)$ (23)+(22)+(20)+(16)+(14)

(26) $\text{DD}_\alpha(q,p)$ and $\text{DD}_\beta(F(q),F(p))$ (23)+(21)+(20)+(17)+Def. 5.1-9

(27) $(p,\alpha)\rho(F(q),\beta)$ (24)+(25)+(26)+Def. 5.1-10

(28) $(p,\alpha)\rho(F(p),\beta)$ (27)+(25)+(26)+Def. 5.1-10

Since (15) leads to a contradiction between (16) and (28), (15) is false.

Thus the second part of the theorem is proven.

$\triangle$

With these preliminary results, the three parts of the induction step in the equivalence proof are now presented as separate lemmas.

Lemma 6.3-1 Given an expansion (Int, J), where $\text{Int} = (\text{St}, I, \text{IE})$, which satisfies the first four Determinacy Axioms, let αf and β be any two computations in any job $J \epsilon J$ such that β is transfer-inclusive of αf. If β is inclusive of α under the natural pointer correspondence $F = F_{\alpha,\beta}$, then β is NPE-inclusive of αf.

Proof:

(1) Let $s = \text{Src}(e,i)$ be the source in $T(f)$. Then $v = V(f)$ is the value of s in αf Def. 4.2-6

(2) There is an entry g in β with $T(g) = T(f)$ Def. 6.3-2

(3) $V(g)$ is the value of s in β (1)+(2)+Def. 4.2-6

(4) $e\epsilon\text{IE} \Rightarrow V(f)$ in αf is not a pointer iff $V(g)$ in β is not a pointer, and if they are not pointers, they are the same   (1)+(3)+Ax. 6.2-4

(5) J is a job for Int   Def. 4.2-2

(6) α, αf, and β are all causal computations for Int

     (5)+Axioms 6.2-2+6.2-1+Def. 4.2-3

(7) e is initiated in α (1)+(6)+Def. 4.2-7

(8) Letting $e = \text{Ex}(d,k)$, there are $\text{In}(I(d))$ input entries to e in α

     (6)+(7)+Def. 4.2-6

(9) There are at most $\text{In}(I(d))$ input entries to e in αf and in β

     (7)+Def. 4.2-6

(10) For any j, there is an entry $\text{Ent}(e,j)$ in α iff there is an entry $\text{Ent}(e,j)$ in β, and if so, those entries have the same transfer

     (8)+(9)+Defs. 4.2-6+6.3-2

(11) For every j, there is an Ent(e,j) in αf iff there is an Ent(e,j) in

β, and if so, their values are either both pointers or both the

same non-pointer   (8)+(9)+(10)+Defs. 6.3-6+6.3-3

(12) I(d) is a pI operation $\Rightarrow$ there is a j such that, in all computations

in which e has the same set of non-pointer-valued input entries

as in αf, the value of the output entries of e equals V(Ent(e,j))

in that computation   Def. 5.1-2

(13) $\Rightarrow$ V(f) = V(Ent$_{\alpha f}$(e,j)) and V(g) = V(Ent$_\beta$(e,j))   (11)+(1)+(3)

(14) $\Rightarrow$ V(f) is not a pointer iff V(g) is not a pointer, and if they are

not pointers, then they are the same   (11)

(15) e is not in IE and I(d) is not a pI or a structure operation $\Rightarrow$ I(d)

is a deterministic action   Ax. 6.2-3

(16) $\wedge$ e's input and output entries have non-pointer values   Const. 5.1-1

(17) $\Rightarrow$ for every j, V(Ent$_{\alpha f}$(e,j)) = V(Ent$_\beta$(e,j))   (11)

(18) $\Rightarrow$ V(g) and V(f) are equal non-pointers   (11)+(15)+(16)+Def. 6.2-1

(19) For any structure operation execution e' initiated before the last

entry in αf, i.e., in α, and any Assign, Update, or Delete

execution A, e' is in reach R(A) in β iff e'$\in$R(A) in α

           Defs. 6.3-3+6.3-5

(20) iff e'$\in$R(A) in αf (6)+(7)+Cor. 6.3-1

(21) If there is an Ent(A,2) in αf, its value is not a pointer

           (19)+Const. 5.1-1

(22) e'$\in$R(A) in αf $\Rightarrow$ e'$\in$R(A) in α $\Rightarrow$ A is initiated in α

          (19)+(20)+(6)+(7)+Lemma 5.3-8

(23) $\Rightarrow$ V(Ent$_{\alpha f}$(A,2)) = V(Ent$_\beta$(A,2))   (7)-(11)+(21)

(24) I(d) is a structure operation $\Rightarrow$ Ent$_\alpha$(e,1) and Ent$_\beta$(e,1) are both

pointer-valued, and have the same transfer, and if I(d) is

anything but Fetch, First, or Copy, e has the same non-pointer

selector input in αf and β (10)+(11)+Const. 5.1-1

(25) $\Rightarrow$ for p = V(Ent$_{\alpha f}$(e,1)) = V(Ent$_\alpha$(e,1)) and p' = V(Ent$_\beta$(e,1)),

F(p) is defined and equal to p'   Defs. 6.3-6+6.3-4

(26) $\Rightarrow$ (p,α)ρ(p',β)   (6)+Def. 6.3-6+Thm. 6.3-1

(27) $\Rightarrow$ (p,αf)ρ(p',β)   (6)+(19)+(20)+Lemma 5.3-10

(28) I(d) is a First or Next $\Rightarrow$ e is in the reach of an Update (Delete)

execution with selector input s in αf iff e is in the reach of an

Update (Delete) execution with selector input s in β

     (7)+(19)+(20)+(22)+(23)

(29) $\Rightarrow$ for i=1,2, Src(e,i) has the same value in αf as in β

     (6)+(24)+(25)+(27)+Const. 5.1-6

(30) $\Rightarrow$ V(f) = V(g)   (1)+(3)

(31) I(d) is an Assign, Update, or Delete and i = 1 $\Rightarrow$ [e is in a reach

in αf $\Rightarrow$ V(f) = V(g) = 0]   (1)+(3)+Consts. 5.1-3+5.1-4

(32) $\wedge$ [e is not in a reach in αf $\Rightarrow$ V(f) = V(g)]

     (6)+(24)+(25)+(27)+(1)+(3)+Const. 5.1-5

(33) I(d) is a Fetch or Assign and e$\in$R(A) in αf $\Rightarrow$ e$\in$R(A) in β

     (7)+(19)+(20)

(34) $\Rightarrow$ letting v be V(Ent$_{\alpha f}$(A,2)) = V(Ent$_\beta$(A,2)), [i = 1 and I(d) is

Fetch $\Rightarrow$ V(f) = V(g) = v] $\wedge$ [i = 2 $\Rightarrow$ V(f) = V(g) = (v$\neq$nil)]

     (22)+(23)+Const. 5.1-3

(35) I(d) is Fetch or Assign and e is in no reach in αf $\Rightarrow$ V(g) = V(f)

     (6)+(24)+(25)+(27)+(1)+(3)+Const. 5.1-5

(36) I(d) is Select, Update, or Delete and i = 2 ⇒ [e is in the reach

of an Update in αf ⇒ e is in the reach of an Update in α ⇒

$V(f) = V(g) = $ <u>true</u>] ∧ [e is in the reach of a Delete in αf ⇒

$V(f) = V(g) = $ <u>false</u>]   (7)+(19)+(20)+Const. 5.1-4

(37) ∧ [e is in no reach in αf ⇒ $V(f) = V(g)$]

(6)+(24)+(25)+(27)+(1)+(3)+Const. 5.1-5

(38) I(d) is a Select and i = 1, or I(d) is a Copy ⇒ $V(f)$ and $V(g)$ are

pointers   Const. 5.1-1

(39) $V(f)$ is not a pointer iff $V(g)$ is not a pointer, and if they are

not pointers, they are the same   (4)+(12)+(14)+(15)+(18)+(28)-(38)

(40) β is NPE-inclusive of αf   (39)+Def. 6.3-3

△

Lemma 6.3-2 Given an expansion (Int,J), where Int = (St,I,IE), which

satisfies the first four Determinacy Axioms, let αf and β be any two

computations in any J∈J such that β is transfer-inclusive of αf. If β is

inclusive of α under the natural pointer correspondence $F_{\alpha,\beta}$, then β is

PE-inclusive of αf under $F_{\alpha f,\beta}$.

Proof: Abbreviate $F_{\alpha,\beta}$ as F and $F_{\alpha f,\beta}$ as F'.

(1) Let s = Src(e,i) be the source in T(f). Then $V(f)$ is the value of

s in αf   Def. 4.2-6

(2) There is an entry g in β with $T(g) = T(f)$ Def. 6.3-2

(3) $V(g)$ is the value of s in β   (1)+(2)+Def. 4.2-6

(4) J is a job for Int   Def. 4.2-2

(5) α, αf, and β are all causal computations for Int

(4)+Axioms 6.2-1+6.2-2+Def. 4.2-3

(6) e is initiated in α   (1)+(5)+Def. 4.2-7

(7) Letting e = Ex(d,k), there are exactly In(I(d)) input entries to

e in α, and at most In(I(d)) input entries to e in αf and in β

(5)+(6)+Def. 4.2-6

(8) For any j, there is an entry Ent(e,j) in α iff there is an entry

Ent(e,j) in β, and if so, those entries have the same transfer

(7)+Defs. 4.2-6+6.3-2

(9) For every j, there is an Ent(e,j) in αf iff there is an Ent(e,j) in

β, and if so, their values are either both pointers or both the

same non-pointer   (8)+Defs. 6.3-6+6.3-3

(10) I(d) is a pI operation $\Rightarrow$ there is a j such that, in all computations

in which e has the same set of non-pointer-valued input entries

as in αf, the value of the output entries of e equals V(Ent(e,j))

in that computation   Def. 5.1-2

(11) $\Rightarrow$ V(f) = V(Ent$_{\alpha f}$(e,j)) and V(g) = V(Ent$_\beta$(e,j))   (9)+(1)+(3)

(12) For every h in αf and k in β such that T(k) = T(h), V(k) is a

pointer iff V(h) is a pointer   Lemma 6.3-1+Def. 6.3-3

(13) and if h is in α, and those values are pointers, then F(V(h)) is

defined and equal to V(k)   Defs. 6.3-6+6.3-4

(14) For each pointer r which is the value of an entry in α, r is the

value in α of the output entries of a Copy execution C $\Rightarrow$ r is the

value in αf of the output entries of C and F(r) is the value in β

of the output entries of C $\Rightarrow$ F'(r) is the value in β of the output

entries of C $\Rightarrow$ F'(r) = F(r)   Def. 6.3-7

(15) r is the value of an output entry of a Copy execution in αf $\Rightarrow$ the

first entry with a value of r in αf, which is in α, is an output

entry of C   (5)+(14)+Lemma 5.3-8

(16) For any structure operation execution e' initiated in α, and for

any Assign, Update, or Delete execution A, $e' \in R(A)$ in β iff

$e' \in R(A)$ in α Defs. 6.3-6+6.3-5

(17) iff $e' \in R(A)$ in αf (5)+(6)+Cor. 6.3-1

(18) r is not the value in α of the output entries of a Copy execution

$\Rightarrow F(r) = r'$, where r' is not the value in β of the output entries

of a Copy execution and $(r,\alpha)\rho(r',\beta)$ (13)+Def. 6.3-7

(19) $\wedge$ r is not the value of an output entry of a Copy execution in αf

                                    (15)

(20) $\Rightarrow (r,\alpha f)\rho(r',\beta)$ (5)+(16)+(17)+Lemma 5.3-10

(21) $\Rightarrow F'(r) = r' = F(r)$ (19)+(18)+Def. 6.3-7

(22) Assume $V(f)$ is a pointer p. Then $V(g)$ is a pointer p'   (12)+(2)

(23) e either is in IE or is a pI execution, a Copy, or a Select

execution   (22)+(1)+Const. 5.1-1

(24) $e \in IE \Rightarrow$ there is a source s = Src(e,i) such that p (p') is the

value of s in αf (β)   (1)+(3)+(22)

(25) $\Rightarrow (p,\alpha f)\rho(p',\beta)$ Def. 5.1-10

(26) $\wedge$ p (p') is not the value of the output entries of a Copy execution

in αf (β)   Const. 5.1-7

(27) $\Rightarrow p' = F'(p)$ Def. 6.3-7

(28) e is a Copy execution $\Rightarrow$ p (p') is the value of the output entries

of a Copy execution C in αf (β)   (1)+(3)+(22)

(29) $\Rightarrow p' = F'(p)$ Def. 6.3-7

(30) e is a pI execution $\Rightarrow \exists j: V(f) = V(Ent_{\alpha f}(e,j)) = p$ and

$V(g) = V(Ent_{\beta}(e,j)) = p'$ (10)+(11)+(22)

(31) $Ent_{\alpha f}(e,j)$ is in α (7)

(32) e is a pI execution $\Rightarrow$ p' = F(p) $\qquad$ (8)+(12)+(13)+(30)+(31)

(33) e is a Select execution which is in a reach in $\alpha f \Rightarrow$ e is in the

$\qquad$ reach of an Update execution U in αf $\qquad$ (22)+Const. 5.1-4

(34) $\Rightarrow$ e is in the reach of U in β $\qquad$ (6)+(16)+(17)

(35) $\wedge$ U is initiated before e, i.e., in α $\qquad$ (5)+(6)+Lemma 5.3-8

(36) $\Rightarrow$ p = V(f) = V($\text{Ent}_{\alpha f}$(U,3)) $\wedge$ p' = V(g) = V(Ent_{β}(U,3))

$\qquad$ (33)+(34)+(1)+(3)+Const. 5.1-4

(37) $\wedge$ $\text{Ent}_{\alpha f}$(U,3) is in α $\qquad$ Def. 4.2-6

(38) $\Rightarrow$ p' = F(p)

(39) e is a Select execution which is in no reach in $\alpha f \Rightarrow$ e is a

$\qquad$ Select execution which is in no reach in β $\qquad$ (6)+(16)+(17)

(40) $\wedge$ V($\text{Ent}_{\alpha f}$(e,2)) and V(Ent_{β}(e,2)) are not pointers and so are

$\qquad$ the same $\qquad$ (9)+Const. 5.1-1

(41) $\wedge$ letting q = V($\text{Ent}_{\alpha f}$(e,1)) and q' = V(Ent_{β}(e,1)), F(q) is defined

$\qquad$ and equal to q' $\qquad$ (6)+(7)+(11)+(12)

(42) $\Rightarrow$ (q,α)ρ(q',β) $\qquad$ (5)+(7)+Def. 6.3-6+Thm. 6.3-1

(43) $\Rightarrow$ (q,αf)ρ(q',β) $\qquad$ (5)+(16)+(17)+Lemma 5.3-10

(44) $\Rightarrow$ (p,αf)ρ(p',β) $\qquad$ (22)+(1)+(3)+(39)+(40)+(41)+Def. 5.1-10

(45) $\wedge$ p (p') is not the value of the output entries of a Copy execution

$\qquad$ in αf (β) $\qquad$ (22)+(1)+(3)+(39)+Const. 5.1-7

(46) $\Rightarrow$ p' = F'(p) $\qquad$ Def. 6.3-7

(47) p' = F'(p)

$\qquad$ (23)+(24)+(27)+(28)+(29)+(39)+(46)+(32)+(33)+(38)+(14)+(18)+(21)

(48) β is PE-inclusive of αf under F' $\qquad$ (12)+(13)+(22)+(47)+Def. 6.3-4

$\triangle$

Lemma 6.3-3 Given an expansion (Int,J), where Int = (St, I, IE), which satisfies the first four Determinacy Axioms, let αf and β be any two computations in any J$\in J$ such that β is transfer-inclusive of αf. If

a. β is NPE-inclusive of αf,

b. β is PE-inclusive of αf under natural pointer correspondence $F_{\alpha f,\beta}$,

c. β is reach-inclusive of α, and

d. β preserves the order of dependent accesses of αf,

then β is reach-inclusive of αf.

Proof: Abbreviate $F_{\alpha f,\beta}$ as F.

(1) J is a job for Int   Def. 4.2-2

(2) αf and β are causal computations for Int   (1)+Ax. 6.2-1+Def. 4.2-3

(3) For every structure operation execution e initiated in α, and for

     any Assign, Update, or Delete execution A, e is in reach R(A) in

     α iff e is in R(A) in β Def. 6.3-5

(4) and e is in R(A) in α iff e is in R(A) in αf (2)+Cor. 6.3-1

(5) If f does not initiate a structure operation execution, then for

     every structure operation execution e initiated in αf, e is

     initiated in α Def. 4.2-6

(6) $\Rightarrow$ e is in R(A) in αf iff e is in R(A) in β (3)+(4)

(7) For any structure operation execution e = Ex(d,k) initiated in αf,

     there are In(I(d)) input entries to e in αf Def. 4.2-6

(8) and e is initiated in β Defs. 6.3-2+4.2-6

(9) In(I(d)) > 1 $\Rightarrow$ V(Ent$_{\alpha f}$(e,2)) is not a pointer   Const. 5.1-1

(10) $\Rightarrow$ V(Ent$_{\beta}$(e,2)) = V(Ent$_{\alpha f}$(e,2))   (5)+Def. 6.3-3

(11) Assume f initiates a structure operation execution e = Ex(d,k)

in αf. For any Assign, Update, or Delete execution A, e∈R(A) in

αf ⇒ A is initiated before e, i.e., in α ⇒

$$V(Ent_{αf}(e,2)) = V(Ent_{β}(e,2)) \qquad (2)+(7)+(9)+(10)+Lemma\ 5.3\text{-}8$$

(12) e is in R(A) in αf iff e and A are executions of one of a few

prescribed combinations of operations. [A is an Update or Delete ∧

e is a Select, Update, or Delete ⇒ $V(Ent_{αf}(e,2)) = V(Ent_{αf}(A,2))$],

and $Ent_{αf}(e,1)$ is in duration D(A) in αf $\qquad$ Defs. 5.1-6+5.1-8

(13) If A is an Update or Delete and e is a Select, Update, or Delete,

then e∈R(A) in αf ⇒ $V(Ent_{αf}(e,2)) = V(Ent_{αf}(A,2))$ iff

$$V(Ent_{β}(e,2)) = V(Ent_{β}(A,2)) \qquad (11)+(7)+(9)+(10)+(8)$$

(14) For every structure operation execution A initiated in αf and every

pointer p, $Ent_{αf}(A,1)$ is in $H_{p}^{αf}$ iff $V(Ent_{αf}(A,1)) = p$ $\qquad$ Def. 5.1-4

(15) iff A is initiated in β and $V(Ent_{β}(A,1)) = F(p)$ $\qquad$ (7)+(8)+Def. 6.3-4

(16) iff $Ent_{β}(A,1)$ is in $H_{F(p)}^{β}$ $\qquad$ Def. 5.1-4

(17) For any pointer p and any computation ω, denote by APS(p,ω) a

sequence of the entries in the set $\{Ent_{ω}(A,1) \mid$ A is an Assign$\}$,

arranged in the same relative order as they appear in $H_{p}^{ω}$. Then

every entry in APS(p,αf) is in APS(F(p),β) $\qquad$ (14)+(16)+Def. 5.1-4

Now prove the following:

A: For any p and for any i less than or equal to the length of APS(p,αf),

the ith element in the sequence APS(p,αf) is $Ent_{αf}(A,1)$ iff the ith

element in APS(F(p),β) is $Ent_{β}(A,1)$.

Proof is by induction on i.

Basis: i = 1.

(18) Let $Ent_{αf}(A,1)$ be the first element in APS(p,αf). $Ent_{β}(A,1)$ is

not the first element in APS(F(p),β) ⇒ it is the jth, j > 1 $\qquad$ (17)

(19) → letting $Ent_\beta(A',1)$ be the $j-1^{st}$ element in $APS(F(p),\beta)$, there is

no Assign execution input entry between $Ent_\beta(A',1)$ and $Ent_\beta(A,1)$

in $H^\beta_{F(p)}$ (17)

(20) → $Ent_\beta(A,1)$ is in the duration $D(A')$ in β Def. 5.1-5

(21) → A is in $R(A')$ in β Def. 5.1-6

(22) → A' initiates before A in αf (19)+Def. 6.3-1

(23) → $Ent_{\alpha f}(A',1)$ is in $H^{\alpha f}_p$ (19)+(14)+(16)

(24) → $Ent_{\alpha f}(A',1)$ precedes $Ent_{\alpha f}(A,1)$ in $H^{\alpha f}_p$ (22)+(18)+Def. 5.1-4

(25) → $Ent_{\alpha f}(A,1)$ is not the first element in $APS(p,\alpha f)$ (17)

(26) $Ent_\beta(A,1)$ is the first element in $APS(F(p),\beta)$ (18)+(25)

Induction step: Assume that for some $n > 0$, for all $i \leq n$, $Ent_{\alpha f}(A,1)$

is the ith element of $APS(p,\alpha f)$ iff $Ent_\beta(A,1)$ is the ith element of

$APS(F(p),\beta)$. Consider A such that

(27) $Ent_{\alpha f}(A,1)$ is the $n+1^{st}$ element of $APS(p,\alpha f)$

(28) $\exists j$: $Ent_\beta(A,1)$ is the jth element of $APS(F(p),\beta)$ (17)

(29) $j \leq n$ → $Ent_{\alpha f}(A,1)$ is the jth element of $APS(p,\alpha f)$ ind. hyp.

(30) $j > n$ (27)+(29)

(31) Let A' be such that $Ent_\beta(A',1)$ is the $j-1^{st}$ element of $APS(F(p),\beta)$.

Then $Ent_{\alpha f}(A',1)$ precedes $Ent_{\alpha f}(A,1)$ in $APS(p,\alpha f)$ (19)-(24)

(32) $Ent_{\alpha f}(A',1)$ is the kth element in $APS(p,\alpha f)$ for $k \leq n$ (31)+(27)

(33) $Ent_\beta(A',1)$ is the kth element in $APS(F(p),\beta)$ (32)+ind. hyp.

(34) $j-1 \leq n$; i.e., $j \leq n+1$ (31)+(33)+(32)

(35) $Ent_\beta(A,1)$ is the $n+1^{st}$ element of $APS(F(p),\beta)$ (28)+(30)+(34)

Thus A is proven by induction. For any structure operation execution e

initiated in αf, and any Assign execution A, there are three cases.

Case I: e is in R(A) in αf, $Ent_{\alpha f}(e,1)$ is in the duration D(A) in αf, and $Ent_{\alpha f}(e,1)$ and $Ent_{\alpha f}(A,1)$ are in the same access history $H_p^{\alpha f}$.

(36) e and A are executions of one of the right combinations of

operations, and if A is an Update or Delete and e is a Select,

Update, or Delete, then $V(Ent_\beta(e,2)) = V(Ent_\beta(A,2))$ (12)+(13)

(37) $Ent_\beta(e,1)$ and $Ent_\beta(A,1)$ are both in $H_{F(p)}^\beta$ (11)+(14)+(16)

(38) Letting i be such that $Ent_{\alpha f}(A,1)$ is the ith element of $APS(p,\alpha f)$,

$Ent_{\alpha f}(e,1)$ follows $Ent_{\alpha f}(A,1)$ in $H_p^{\alpha f}$, but does not follow in $H_p^{\alpha f}$

the $i+1^{st}$ element of $APS(p,\alpha f)$ (11)+(17)+Def. 5.1-5

(39) A initiates before e in α Def. 6.3-1

(40) $Ent_\beta(e,1)$ follows $Ent_\beta(A,1)$ in $H_{F(p)}^\beta$ (37)+(39)+Def. 5.1-4

(41) $Ent_\beta(A,1)$ is the ith element of $APS(F(p),\beta)$ (38)+A

(42) $e \notin R(A)$ in β $\Rightarrow$ there is an Assign execution $A' \neq A$ such that $Ent_\beta(e,1)$

is in $D(A')$ in β and $e \in R(A')$ in β (36)+(40)+Def. 5.1-6

(43) $\Rightarrow Ent_\beta(A',1)$ is between $Ent_\beta(A,1)$ and $Ent_\beta(e,1)$ in $H_{F(p)}^\beta$ Def. 5.1-5

(44) $\wedge$ A' initiates before e in αf (11)+Def. 6.3-1

(45) $\Rightarrow Ent_{\alpha f}(A',1)$ precedes $Ent_{\alpha f}(e,1)$ in $H_p^{\alpha f}$ (43)+(14)+(16)+Def. 5.1-4

(46) $\Rightarrow Ent_{\alpha f}(A',1)$ is the jth element of $APS(p,\alpha f)$ for $j < i$ (38)+(42)

(47) $\Rightarrow Ent_\beta(A',1)$ is the jth element of $APS(F(p),\beta)$ and $j < i$ (38)+A

(48) $\Rightarrow Ent_\beta(A',1)$ precedes $Ent_\beta(A,1)$ in $H_{F(p)}^\beta$ (40)+(41)+(17)

(49) $e \in R(A)$ in β (42)+(43)+(48)

Case II: $e \in R(A)$ and $Ent_{\alpha f}(e,1) \in D(A)$ in αf, $Ent_{\alpha f}(e,1)$ is in $H_p^{\alpha f}$, but $Ent_{\alpha f}(A,1)$ is not in $H_p^{\alpha f}$.

(50) e and A are executions of one of the right combinations of

operations, and if A is an Update or Delete and e is a Select,

Update, or Delete, then $V(Ent_\beta(e,2)) = V(Ent_\beta(A,2))$ (12)+(13)

(51) $\text{Ent}_\beta(e,1)$ is in $H^\beta_{F(p)}$, but $\text{Ent}_\beta(A,1)$ is not   (11)+(14)+(16)

(52) $\text{Ent}_{\alpha f}(e,1)$ precedes in $H^{\alpha f}_p$ the first element of APS(p,αf), p is the

     value of the output entries of some Copy execution C in αf, and

     $\text{Ent}_{\alpha f}(C,1)$ is in D(A) in αf (17)+Def. 5.1-5

(53) C is in R(A) in αf (52)+Def. 5.1-6

(54) $\text{Ent}_{\alpha f}(e,1)$ has value p and is in αf Def. 5.1-4

(55) There is an entry in αf which is an output entry of C

                           (2)+(54)+(52)+Lemma 5.3-8

(56) C's initiating entry precedes that entry, i.e., C is initiated in α

                               (55)+(2)+Def. 4.2-7

(57) C is in R(A) in β (53)+(56)+Def. 6.3-5

(58) $\text{Ent}_\beta(C,1)$ is in D(A) in β (57)+Def. 5.1-6

(59) F(p) is the value of the output entries of C in β (52)+Def. 6.3-7

(60) $e{\notin}R(A)$ in $\beta \Rightarrow$ There is some entry in APS(F(p),β) which precedes

     $\text{Ent}_\beta(e,1)$ in $H^\beta_{F(p)}$ (50)+(51)+(59)+(58)+(17)+Def. 5.1-5

(61) $\Rightarrow$ There is some A' such that $\text{Ent}_\beta(A',1){\in}$APS(F(p),$\beta$) and $\text{Ent}_\beta(e,1)$

     is in D(A') in β (17)+Def. 5.1-5

(62) $\Rightarrow$ $e{\in}R(A')$ in β (50)+Def. 5.1-6

(63) $\Rightarrow$ A' initiates before e in αf Def. 6.3-1

(64) $\Rightarrow$ $\text{Ent}_{\alpha f}(A',1)$ is in $H^{\alpha f}_p$ (60)+(14)+(16)

(65) $\Rightarrow$ $\text{Ent}_{\alpha f}(A',1)$, which is in APS(p,$\alpha f$), precedes $\text{Ent}_{\alpha f}(e,1)$ in $H^{\alpha f}_p$

                               (17)+(63)+Def. 5.1-4

(66) $e{\in}R(A)$ in β (60)+(65)+(52)

Case III: There is no Assign execution A such that $e{\in}R(A)$ in αf. Prove

that there is no Assign execution A such that $e{\in}R(A)$ in β by contradiction.

(67) Assume there is an Assign execution A such that $e \in R(A)$ in β

(68) e and A are executions of one of the proper combinations of operations, if A is an Update or Delete and e is a Select, Update, or Delete, then $V(Ent_{\beta}(e,2)) = V(Ent_{\beta}(A,2))$, and $Ent_{\beta}(e,1) \in D(A)$ in β

Def. 5.1-6

(69) If A is an Update or Delete and e is a Select, Update, or Delete, then $V(Ent_{\alpha f}(e,2)) = V(Ent_{\alpha f}(A,2))$ (13)

(70) $Ent_{\alpha f}(e,1) \in D(A)$ in $\alpha f \Rightarrow e \in R(A)$ (68)+(69)+Def. 5.1-6

(71) $Ent_{\alpha f}(e,1) \notin D(A)$ in αf (70)

(72) There is a pointer p such that $Ent_{\alpha f}(e,1)$ precedes in $H_p^{\alpha f}$ the first element of $APS(p,\alpha f)$, and either p is not the value in αf of the output entries of a Copy execution, or p is the value in αf of the output entries of a Copy execution C, but $Ent_{\alpha f}(C,1)$ is not in $D(A)$ in αf (17)+(71)+Def. 5.1-5

(73) There is some entry in $APS(F(p),\beta)$ which precedes $Ent_{\beta}(e,1)$ in $H_{F(p)}^{\beta} \Rightarrow$ there is some A' such that $Ent_{\alpha f}(A',1)$ is in $APS(p,\alpha f)$ and precedes $Ent_{\alpha f}(e,1)$ in $H_p^{\alpha f}$ (69)+(60)+(65)

(74) There is no entry in $APS(F(p),\beta)$ which precedes $Ent_{\beta}(e,1)$ in $H_{F(p)}^{\beta}$

(73)+(72)

(75) $F(p)$ is the value of the output entries of a Copy execution C in β iff p is the value of the output entries of C in αf Def. 6.3-7

(76) p is not the value of the output entries of a Copy execution in αf $\Rightarrow F(p)$ is not the value of the output entries of a Copy execution in β (75)

(77) p is the value of the output entries of a Copy execution C in αf

and $Ent_\beta(C,1)$ is in $D(A)$ in $\beta \Rightarrow C$ is initiated in α (72)+(54)-(56)

(78) $\wedge$ C is in R(A) in β Def. 5.1-6

(79) $\Rightarrow$ C is in R(A) in αf   Def. 6.3-5

(80) $\Rightarrow Ent_{\alpha f}(C,1)$ is in D(A) in αf   Def. 5.1-6

(81) p is the value of the output entries of Copy execution C, but

    $Ent_{\alpha f}(C,1)$ is not in D(A) in αf $\Rightarrow$ F(p) is the value of the output

    entries of C in β (75)

(82) $\wedge$ $Ent_\beta(C,1)$ is not in D(A) in β (77)+(80)

(83) $Ent_\beta(e,1)$ is not in D(A) in β (72)+(74)+(76)+(81)+(82)+Def. 5.1-5

Since (67) implies a contradiction between (68) and (83), (67) is false.

(84) There is no Assign execution A such that e$\in$R(A) in β

(85) If f initiates a structure operation execution e in αf, then for

    any Assign execution A, e$\in$R(A) in αf iff e$\in$R(A) in β (49)+(66)+(84)

Replacing "Assign execution A" with "Update or Delete execution A with

V(Ent(A,2)) = V(Ent(e,2))" in (17) through (85) yields a proof of

(86) For any Update or Delete execution A, e$\in$R(A) in αf iff e$\in$R(A) in β

(87) β is reach-inclusive of αf   (5)+(6)+(11)+(85)+(86)+Def. 6.3-5

$$\triangle$$

   Now the framework of the induction is easily built on these three

lemmas, to complete the proof of the basic requirements for equivalence.

Theorem 6.3-2 Given an expansion (Int,J) which satisfies the first four

Determinacy Axioms, let ω_1 and ω_2 be any two computations in any job J$\in$$J$.

If ω_2 is transfer-congruent to ω_1 and preserves the order of dependent

accesses of ω_1, then ω_2 is equivalent to ω_1.

<u>Proof</u>: First prove, by induction on the lengths of the prefixes of ω_1, that ω_2 is inclusive of ω_1 under the natural pointer correspondence $F = F_{\omega_1,\omega_2}$. Induction hypothesis is that ω_2 is inclusive of a prefix α of ω_1 under F_{α,ω_2}.

Basis: $|\alpha| = 0$

  (1) There are no entries in α, so that there are no executions of

      structure operations initiated in α Def. 4.2-6

  (2) ω_2 is vacuously inclusive of α (1)+Defs. 6.3-2→6.3-6

Induction step: Assume that ω_2 is inclusive of the length-n prefix α of ω_1 under F_{α,ω_2}, and consider the length-n+1 prefix αf of ω_1.

  (3) J is a job for Int   Def. 4.2-2

  (4) ω_1, ω_2, α, and αf are all causal computations for Int and are all

      in J   (3)+Axioms 6.2-1+6.2-2+Def. 4.2-3

  (5) ω_2 is transfer-inclusive of ω_1, hence of αf Def. 6.3-2

  (6) ω_2 is inclusive of α under F_{α,ω_2} ind. hyp.

  (7) ω_2 is NPE-inclusive of αf (4)+(5)+(6)+Lemma 6.3-1

  (8) ω_2 is PE-inclusive of αf under $F_{\alpha f,\omega_2}$ (4)+(5)+(6)+Lemma 6.3-2

  (9) ω_2 is reach-inclusive of α (6)+Def. 6.3-6

 (10) For each structure operation execution e initiated in αf, e is

      initiated in ω_1 Def. 4.2-6

 (11) For each such e, and each Assign, Update, or Delete execution A,

      $e \in R(A)$ in $\alpha f \Rightarrow$ A is initiated before e in αf (4)+Lemma 5.3-8

 (12) $e \in R(A)$ in αf and $Ent_{\alpha f}(A,1)$ and $Ent_{\alpha f}(e,1)$ are in the same access

      history in $\alpha f \Rightarrow e \in R(A)$ in ω_1 (4)+(10)+(11)+Cor. 6.3-1

 (13) $\wedge\ V(Ent_{\alpha f}(e,1)) = V(Ent_{\alpha f}(A,1))$ Def. 5.1-4

(14) $\Rightarrow$ Ent$_{\alpha f}$(e,1) and Ent$_{\alpha f}$(A,1) are in the same access history in ω_1

(10)+(11)+Def. 5.1-4

(15) $\Rightarrow$ A is initiated before e in ω_2 (12)+Def. 6.3-1

(16) eϵR(A) in ω_2 and Ent$_{\omega_2}$(e,1) and Ent$_{\omega_2}$(A,1) are in the same access

history in ω_2 $\Rightarrow$ A is initiated before e in ω_1 (10)+Def. 6.3-1

(17) $\Rightarrow$ A is initiated before e in αf (10)+Def. 4.2-6

(18) ω_2 preserves the order of dependent accesses of αf

(10)+(11)+(12)+(15)+(16)+(17)+Def. 6.3-1

(19) ω_2 is reach-inclusive of αf (4)+(5)+(7)+(8)+(9)+(18)+Lemma 6.3-3

(20) ω_2 is inclusive of αf under F$_{\alpha f, \omega_2}$ (5)+(7)+(8)+(19)+Def. 6.3-6

Thus it is proven inductively that

(21) ω_2 is inclusive of ω_1 under F

(22) The set of transfers of the entries in ω_1 and ω_2 are identical

Def. 6.3-2

(23) For any entry f in ω_1, let g be the entry in ω_2 such that T(g) = T(f)

Then V(f) is not a pointer iff V(g) is not a pointer, and if those

values are not pointers, they are equal   (21)+Defs. 6.3-6+6.3-3

(24) F is one-to-one   (4)+(22)+Def. 6.3-2+Thm. 6.3-1

(25) There is a one-to-one pointer correspondence F such that V(f) is a

pointer iff V(g) is a pointer, and if they are pointers, then

F(V(f)) is defined and equal to V(g)   (21)+(24)+Defs. 6.3-6+6.3-4

(26) For any Assign, Update, or Delete execution A, eϵR(A) in ω_1 $\Rightarrow$

Ent(e,1) is in D(A) in ω_1 Defs. 5.1-6+5.1-8

(27) $\Rightarrow$ Ent(e,1) is in an access history in ω_1 Defs. 5.1-5+5.1-7

(28) $\Rightarrow$ e is initiated in ω_1 Def. 5.1-4

(29) $\Rightarrow$ e$\in$R(A) in ω_2	(26)+(21)+Defs. 6.3-6+6.3-5
(30) e$\in$R(A) in ω_2 $\Rightarrow$ e is initiated in ω_2	(26)-(28)
(31) $\Rightarrow$ e is initiated in ω_1	(22)+Def. 4.2-6
(32) $\Rightarrow$ e$\in$R(A) in ω_1	(30)+(21)+Defs. 6.3-6+6.3-5
(33) R(A) in ω_1 equals R(A) in ω_2	(26)+(29)+(30)+(32)
(34) ω_2 is equivalent to ω_1	(22)+(23)+(25)+(33)+Def. 6.1-1

Q.E.D.

6.4 The Determinacy Proof

This concluding section applies the foregoing general equivalence
result to the immediate problem of proving that the Determinacy Axioms
imply determinacy. The Determinacy Proof Technique introduced in
Chapter 4 *is used to prove, as* required, that any two halted computations
ω_1 and ω_2 in a job J are equivalent. A sequence of computations is
constructed in which the first is ω_1, each succeeding computation is in J,
is halted, and is equivalent to the preceding, and the last computation
is equivalent to ω_2.

Each computation in this sequence is derived from the preceding one
by permuting a single entry zero or more positions to the left. That
such a permutation results in an equivalent computation is proven first,
in two lemmas. The first of these proves that moving an entry one
position to the left yields an equivalent computation which is in the same
job. The second lemma then inductively extends *this to a permutation of*
any number of positions to the left.

Lemma 6.4-1   Let (Int, J), where $Int = (St, I, IE)$, be any expansion from an
S-S model which satisfies the Determinacy Axioms. For any $J \in J$, for any
computation $\alpha g f \beta$ in J such that

(1) $T(f) \in ET_J(\alpha)$

$\alpha f g \beta$ is in J and is equivalent to $\alpha g f \beta$.

Proof:

(2) $\alpha g f$ is in J and so is causal   Axioms 6.2-1+6.2-2+Def. 4.2-7

(3) Assume there is a computation $\alpha \bar{f} \bar{g} \in J$ in which $T(\bar{f}) = T(f)$ and

$T(\bar{g}) = T(g)$. Let e be any structure operation execution initiated

in $\alpha g f$, and let A be any Assign, Update, or Delete execution. Then

$e \in R(A)$ in $\alpha g f$ and $Ent(A,1)$ and $Ent(e,1)$ are in the same access

history in $\alpha g f \Rightarrow$ it is not the case that f and g are the initiating

entries of e and A   (2)+Ax. 6.2-7

(4) $\wedge$ A's initiating entry precedes e's in $\alpha g f$ (2)+Lemma 5.3-8

(5) $\Rightarrow$ A's initiating entry is in α (3)

(6) $\Rightarrow$ A's initiating entry precedes e's in $\alpha \bar{f} \bar{g}$ (4)

By symmetry (exchanging g for $\bar{f}$ and f for $\bar{g}$),

(7) $e \in R(A)$ in $\alpha \bar{f} \bar{g}$ and $Ent(A,1)$ and $Ent(e,1)$ are in the same access

history in $\alpha \bar{f} \bar{g} \Rightarrow$ A's initiating entry precedes e's in $\alpha g f$

(8) If there is an $\alpha \bar{f} \bar{g} \in J$ in which $T(\bar{f}) = T(f)$ and $T(\bar{g}) = T(g)$, then it

preserves the order of dependent accesses of $\alpha g f$

(3)+(6)+(7)+Def. 6.3-1

Now prove the Lemma by induction on the length of β.

Basis: $|\beta| = 0$.

(9) $\alpha g f \beta = \alpha g f$ is in J

(10) $\exists \bar{f}: T(\bar{f}) = T(f)$ and $\alpha \bar{f} \in J$ (1)+Def. 6.2-2

(11) $\alpha g \in J$ (9)+Ax. 6.2-2+Def. 4.2-7

(12) $T(g) \in ET_J(\alpha)$ (11)+Def. 6.2-2

(13) $T(g)$ and $T(f)$ have distinct destinations, so $T(g) \neq T(\bar{f})$

                                                          (9)+(10)+Def. 4.2-6

(14) $T(g) \in ET_J(\alpha \bar{f})$ (10)+(13)+(12)+Ax. 6.2-5

(15) $\exists \bar{g}: T(\bar{g}) = T(g)$ and $\alpha \bar{f}\bar{g} \in J$ (14)+Def. 6.2-2

(16) $\alpha \bar{f}\bar{g}$ is transfer-congruent to αgf (10)+(15)+Def. 6.3-2

(17) $\alpha \bar{f}\bar{g}$ preserves the order of dependent accesses of αgf (10)+(15)+(8)

(18) $\alpha \bar{f}\bar{g}$ is equivalent to αgf (16)+(17)+Thm. 6.3-2

(19) $V(f)$ is not a pointer $\Rightarrow V(\bar{f}) = V(f)$ (10)+(18)+Def. 6.1-1

(20) $\Rightarrow \bar{f} = f$ (10)+Def. 4.2-5

(21) $V(g)$ is not a pointer $\Rightarrow \bar{g} = g$ (15)+(18)+Defs. 6.1-1+4.2-5

If $V(f)$ is a pointer, there are two cases to consider. Case I:

(22) There is an entry $k \in \alpha$ with $V(k) = V(f)$

(23) There is an entry $\bar{k}$ in $\alpha \bar{f}\bar{g}$ with $T(\bar{k}) = T(k)$, and since $k \in \alpha$,

     $V(\bar{k}) = V(k)$ (16)+Def. 6.3-2

(24) There is a one-to-one mapping F over pointers such that

     $V(\bar{k}) = F(V(k))$ and $V(\bar{f}) = F(V(f))$ (10)+(23)+(18)+Def. 6.1-1

(25) $F(V(k)) = F(V(f))$ (24)+(22)

(26) $V(\bar{f}) = F(V(f)) = F(V(k)) = V(\bar{k}) = V(k) = V(f)$ (24)+(25)+(23)+(22)

(27) $\alpha \bar{f}\bar{g}$ is in J   (15)+(26)+(10)

Case II:

(28) There is no entry in α with the same value as f

(29) Let $p = V(\bar{f})$ and $p' = V(f)$. Define the map Y by

$$Y(q) = \begin{cases} q & \text{if } q \neq p \\ p' & \text{if } q = p \end{cases}$$

Then substituting for each entry k in $\alpha\overline{\overline{fg}}$ a similar entry, with

transfer $T(k)$ and value $V(k)$, if that is not a pointer, or $Y(V(k))$

otherwise, yields $\alpha f\overline{g}$ (or αfg if $V(\overline{g}) = p$). Therefore, if $V(\overline{g}) = p$

then $\alpha fg \simeq \alpha\overline{\overline{fg}}$, else $\alpha f\overline{g} \simeq \alpha\overline{\overline{fg}}$ (28)+Def. 5.1-3

(30) If $V(\overline{g}) = p$, then αfg is in J, else $\alpha f\overline{g}$ is in J   (15)+(29)+Const. 5.1-2

By applying the same reasoning as (22)-(30),

(31) if $V(g)$ is a pointer not equal to p', then αfg is in J

(32) αfg is in J   (19)+(20)+(21)+(30)+(31)

(33) αfg is transfer-congruent to αgf Def. 6.3-2

(34) αfg preserves the order of dependent accesses of αgf (32)+(8)

(35) αfg is equivalent to αgf (33)+(34)+Thm. 6.3-2

Induction step: Assume that the Lemma is true for any $\alpha gf\beta$ in which

$|\beta| = n \geq 0$, and consider

(36) $\alpha gf\beta = \alpha gf\delta h$, in which $|\beta| = n+1$

(37) $\alpha gf\delta \epsilon J$, $\alpha gf \epsilon J$, and $T(h) \epsilon ET_J(\alpha gf\delta)$ (36)+Ax. 6.2-2+Defs. 4.2-7+6.2-2

(38) J is a job for Int   Def. 4.2-2

(39) $\alpha gf\delta h$ and αgf are causal computations for Int

                             (37)+(38)+Ax. 6.2-1+Def. 4.2-3

(40) $\alpha fg\delta$ is in J   (36)+(37)+ind. hyp.

(41) $ET_J(\alpha fg\delta) = ET_J(\alpha gf\delta)$ (37)+(40)+Ax. 6.2-6

(42) $T(h)$ is in $ET_J(\alpha fg\delta)$ (37)+(41)

(43) $\exists \overline{h}: T(\overline{h}) = T(h)$ and $\alpha fg\delta\overline{h} \epsilon J$ (42)+Def. 6.2-2

(44) $\alpha fg\delta\overline{h}$ is transfer-congruent to $\alpha gf\delta h$ (43)+Def. 6.3-2

(45) Let $\alpha fg\delta\overline{h}$ be <u>any</u> computation in J in which $T(\overline{h}) = T(h)$. Let e be

any structure operation execution initiated in $\alpha gf\delta h$, and let A
be any Assign, Update, or Delete execution. There are then two
cases to consider.

Case I:

(46) e's initiating entry is in αgf

(47) $e\epsilon R(A)$ in $\alpha gf\delta h \Rightarrow$ A is initiated before e, i.e., in αgf

$$(39)+(46)+\text{Lemma } 5.3-8$$

(48) $e\epsilon R(A)$ in $\alpha gf\delta h$ and $Ent(e,1)$ and $Ent(A,1)$ are in the same access

history in $\alpha gf\delta h \Rightarrow e\epsilon R(A)$ in αgf $\qquad (39)+(46)+\text{Cor. } 6.3-1$

(49) $\wedge$ $V(Ent(e,1)) = V(Ent(A,1))$ $\qquad\qquad\qquad$ Def. 5.1-4

(50) $\Rightarrow Ent(e,1)$ and $Ent(A,1)$ are in the same access history in αgf

$$(46)+(47)+\text{Def. } 5.1-4$$

(51) $\Rightarrow$ A is initiated before e in αfg $\qquad\qquad (34)+(48)+\text{Def. } 6.3-1$

(52) $\Rightarrow$ A is initiated before e in $\alpha fg\delta\bar{h}$ $\qquad\qquad$ Def. 4.2-6

By symmetry

(53) $e\epsilon R(A)$ in $\alpha fg\delta\bar{h}$ and $Ent(e,1)$ and $Ent(A,1)$ are in the same access

history in $\alpha fg\delta\bar{h} \Rightarrow$ A is initiated before e in $\alpha gf\delta h$

(54) $(46) \Rightarrow \alpha fg\delta\bar{h}$ preserves the order of dependent accesses of $\alpha gf\delta h$

$$(48)+(52)+(53)+\text{Def. } 6.3-1$$

Case II:

(55) e's initiating entry is in δh

(56) $e\epsilon R(A)$ in $\alpha gf\delta h \Rightarrow$ A is initiated before e in $\subset gf\delta h$ $\quad(39)+\text{Lemma } 5.3-8$

(57) $\Rightarrow$ letting $A = Ex(d,k)$, $In(/(d))$ input entries to A precede e's

initiating entry in $\alpha gf\delta h$ $\qquad\qquad\qquad$ Def. 4.2-6

(58) $\Rightarrow In(/(d))$ input entries to A precede e's initiating entry in $\alpha fg\delta\bar{h}$

$$(55)$$

(59) $\Rightarrow$ A is initiated before e in $\alpha f g \delta \bar{h}$ Def. 4.2-6

By symmetry

(60) $e \in R(A)$ in $\alpha f g \delta \bar{h} \Rightarrow$ A is initiated before e in $\alpha g f \delta h$

(61) $\alpha f g \delta \bar{h}$ preserves the order of dependent accesses of $\alpha g f \delta h$

$$(45)+(54)+(55)+(59)+(60)+\text{Def. } 6.3\text{-}1$$

(62) $\alpha f g \delta \bar{h}$ is equivalent to $\alpha g f \delta h$ $(44)+(61)+\text{Thm. } 6.3\text{-}2$

By reasoning similar to (19)-(32)

(63) $\alpha f g \delta h$ is in J

(64) $\alpha f g \delta h$ is transfer-congruent to $\alpha g f \delta h$ Def. 6.3-2

(65) $\alpha f g \delta h$ preserves the order of dependent accesses of $\alpha g f \delta h$ $(45)+(61)$

(66) $\alpha f g \beta$ is equivalent to $\alpha g f \beta$ $(36)+(64)+(65)+\text{Thm. } 6.3\text{-}2$

$\triangle$

Lemma 6.4-2 Let $(\text{Int}, \mathcal{J})$ be any expansion from an S-S model satisfying the

Determinacy Axioms. Then for any computation $\alpha \beta f \gamma$ in any $J \in \mathcal{J}$ in which

(1) $T(f) \in ET_J(\alpha)$

$\alpha f \beta \gamma$ is in J, and is equivalent to $\alpha \beta f \gamma$, and $ET_J(\alpha f \beta \gamma) = ET_J(\alpha \beta f \gamma)$.

Proof: By induction on $|\beta|$.

Basis: $|\beta| = 0$. Then $\alpha f \beta \gamma = \alpha \beta f \gamma$, so the Lemma is trivially true.

Induction step: Assume the Lemma is true for an $\alpha \beta f \gamma \in J$ in which

$|\beta| = n \geq 0$, and consider any

(2) $\alpha \beta f \gamma = \alpha g \delta f \gamma$ in J in which $|\beta| = n+1$

(3) $T(f) \neq T(g)$ Def. 4.2-6

(4) $\alpha g \in J$ Ax. 6.2-2+Def. 4.2-7

(5) $T(f) \in ET_J(\alpha g)$ $(1)+(3)+(4)+\text{Ax. } 6.2\text{-}5$

(6) $\alpha gf\delta\gamma$ is in J and is equivalent to $\alpha g\delta f\gamma$, and

$$ET_J(\alpha gf\delta\gamma) = ET_J(\alpha g\delta f\gamma) \qquad\qquad (5)+(2)+\text{ind. hyp.}$$

(7) $\alpha fg\delta\gamma$ is in J and is equivalent to $\alpha gf\delta\gamma$ $\qquad$ (1)+(6)+Lemma 6.4-1

(8) $\alpha fg\delta\gamma$ is equivalent to $\alpha g\delta f\gamma$ $\qquad\qquad$ (6)+(7)+Def. 6.1-1

(9) $ET_J(\alpha fg\delta\gamma) = ET_J(\alpha gf\delta\gamma)$ $\qquad\qquad$ (6)+(7)+Ax. 6.2-6

(10) $ET_J(\alpha fg\delta\gamma) = ET_J(\alpha g\delta f\gamma)$ $\qquad\qquad\qquad$ (6)+(9)

$$\triangle$$

Now finally the Determinacy Proof Technique is easily utilized to produce the following quite general result:

Theorem 6.4-1 Every expansion (Int,J) from an S-S model which satisfies the Determinacy Axioms is determinate.

Proof: It is required to prove for any $J \epsilon J$ that, by virtue of satisfying the Determinacy Axioms, any two halted computations ω and ω' in J are equivalent. Assume without loss of generality that

(1) $|\omega| \le |\omega'| = n$

Inductively construct from ω a sequence of computations $\omega_0,\omega_1,\ldots,\omega_n$, in which each ω_k can be written as

(2) $\omega_k = \bar{\alpha}_k\beta_k$, where

> letting α_k be the length-k prefix of ω', for $i=1,\ldots,k$, the ith entry in $\bar{\alpha}_k$ has the same transfer as the ith entry in α_k, and
>
> $\beta_k = \omega - \bar{\alpha}_k$; i.e., β_k is derived from ω by striking out every entry which has the same transfer as an entry in $\bar{\alpha}_k$

Prove, by induction on k, the following hypotheses:

A: ω_k is in J

B: ω_k is halted in J

C: ω_k is equivalent to ω

D: $\bar{\alpha}_k$ is equivalent to α_k

Basis: $k = 0$. Then $\alpha_k = \lambda$, so $\bar{\alpha}_k = \lambda$. Then $\beta_k = \omega - \lambda = \omega$. So

   $\omega = \alpha_0 \beta_0 = \omega_0$, and A, B, C, and D are trivially true.

Induction step: For any k, $0 \leq k < n$, assume that there is an $\omega_k = \bar{\alpha}_k \beta_k$

such that A, B, C, and D are true of ω_k, and construct ω_{k+1}.

 (3) Let f be the $k+1^{st}$ entry in ω', i.e., $\alpha_{k+1} = \alpha_k f$ is a prefix of ω',

   hence is in J   (2)+Ax. 6.2-2+Def. 4.2-7

 (4) $T(f) \in ET_J(\alpha_k)$ (3)+Def. 6.2-2

 (5) $\bar{\alpha}_k$ is equivalent to α_k ind. hyp. D

 (6) The ith entry in $\bar{\alpha}_k$ has the same transfer as the ith entry in α_k (2)

 (7) If the value of the ith entry in $\bar{\alpha}_k$ is non-pointer v, then the value

   of the ith entry in α_k is non-pointer v   (5)+(6)+Def. 6.1-1

 (8) There is a one-to-one pointer correspondence F such that if the

   value of the ith entry in $\bar{\alpha}_k$ is pointer p, then the value of the

   i^{th} entry in α_k is F(p)   (5)+(6)+Def. 6.1-1

 (9) Define Y: $V_p \rightarrow V_p$ to be

   if $V(f)$ is not a pointer or $F(V(f))$ is defined, then $Y = F$,

   else for all $q \in V_p$,

$$Y(q) = \begin{cases} F(q) & \text{if that is defined} \\ p & \text{where p is not in the range of F, if } q=V(f) \\ \text{undefined otherwise} \end{cases}$$

   Then Y is a one-to-one map over pointers   (8)

(10) Let $\bar{\bar{f}}$ be an entry with $T(\bar{\bar{f}}) = T(f)$ and $V(f)$ equals, if $V(f)$ is not a pointer, then $V(f)$, else $Y(V(f))$. Then $\bar{\alpha}_k \bar{\bar{f}}$ can be derived from $\alpha_k f$ by replacing each entry $g \in \alpha_k f$ with a similar entry whose transfer is $T(g)$ and whose value is, if $V(g)$ is non-pointer, then $V(g)$, else $Y(V(g))$ (6)+(7)+(8)+(9)

(11) $\bar{\alpha}_k \bar{\bar{f}} \simeq \alpha_k f$ (10)+(9)+Def. 5.1-3

(12) $\bar{\alpha}_k \bar{\bar{f}}$ is in J   (3)+(11)+Const. 5.1-2

(13) $T(f) = T(\bar{\bar{f}})$ is in $ET_J(\bar{\alpha}_k)$ (10)+(12)+Def. 6.2-2

(14) $\omega_k = \bar{\alpha}_k \beta_k$ is in J   (2)+ind. hyp. A

(15) $\bar{\alpha}_k$ is in J   (14)+Ax. 6.2-2+Def. 4.2-7

(16) $[\exists \bar{\bar{f}} \in \beta_k : T(\bar{\bar{f}}) = T(f)]$ or $[T(f) \in ET_J(\bar{\alpha}_k \beta_k)]$ (15)+(13)+(14)+Lemma 6.2-1

(17) $T(f) \in ET_J(\bar{\alpha}_k \beta_k) \Rightarrow \exists h : \bar{\alpha}_k \beta_k h \in J$ Def. 6.2-2

(18) $\Rightarrow \bar{\alpha}_k \beta_k = \omega_k$ is a proper prefix of a computation in J, and so is not halted in J   Def. 4.2-7

(19) $T(f) \notin ET_J(\bar{\alpha}_k \beta_k)$ (17)+(18)+ind. hyp. B

(20) $\exists \bar{\bar{f}} \in \beta_k : T(\bar{\bar{f}}) = T(f)$ (16)+(19)

(21) ω_k may be written as $\omega_k = \bar{\alpha}_k \gamma \bar{\bar{f}} \delta$ (14)+(20)

(22) $T(\bar{\bar{f}}) \in ET_J(\bar{\alpha}_k)$ (13)+(20)

(23) $\bar{\alpha}_k \bar{\bar{f}} \gamma \delta$ is in J and is equivalent to $\bar{\alpha}_k \gamma \bar{\bar{f}} \delta$, and $ET_J(\bar{\alpha}_k \bar{\bar{f}} \gamma \delta) = ET_J(\bar{\alpha}_k \gamma \bar{\bar{f}} \delta)$ (21)+(14)+(22)+Lemma 6.4-2

(24) Let $\bar{\alpha}_{k+1} = \bar{\alpha}_k \bar{\bar{f}}$ (2)+(3)+(20)

(25) $\gamma \delta = \beta_k - \bar{\bar{f}} = \omega - \bar{\alpha}_k - \bar{\bar{f}} = \omega - \bar{\alpha}_k \bar{\bar{f}} = \omega - \bar{\alpha}_{k+1}$ (21)+(14)+(2)+(24)

(26) Letting $\beta_{k+1} = \gamma \delta$, there is an $\omega_{k+1} = \bar{\alpha}_{k+1} \beta_{k+1} = \bar{\alpha}_k \bar{\bar{f}} \gamma \delta$ in J, which is equivalent to $\bar{\alpha}_k \gamma \bar{\bar{f}} \delta = \omega_k$, and $ET_J(\omega_{k+1}) = ET_J(\omega_k)$

       (25)+(2)+(24)+(23)

(27) A for ω_{k+1} (26)

(28) C for ω_{k+1} (26)+ind. hyp. C

(29) ω_{k+1} is not halted $\rightarrow \exists g: \omega_{k+1} g \in J$ Def. 4.2-7

(30) $\rightarrow T(g) \in T_J(\omega_{k+1})$ Def. 6.2-2

(31) $\rightarrow T(g) \in T_J(\omega_k)$ (26)

(32) $\rightarrow \exists \bar{g}: \omega_k \bar{g} \in J$ Def. 6.2-2

(33) $\rightarrow \omega_k$ is not halted in J   Def. 4.2-7

(34) B for ω_{k+1} (29)+(33)+ind. hyp. B

(35) $\bar{\alpha}_{k+1}$ is transfer-congruent to α_{k+1} (2)+Def. 6.3-2

(36) For any structure operation execution e initiated in α_{k+1} and any

   Assign, Update, or Delete execution A, $e \in R(A)$ in $\alpha_{k+1} \rightarrow$ A is

   initiated before e in α_{k+1} (3)+Ax. 6.2-1+Lemma 5.3-8

(37) $\rightarrow$ A is initiated before e in $\bar{\alpha}_{k+1}$ (2)+Def. 4.2-6

(38) $e \in R(A)$ in $\bar{\alpha}_{k+1} \rightarrow$ A is initiated before e in α_{k+1}

          (24)+(23)+(2)+Axioms 6.2-2+6.2-1+Lemma 5.3-8+Def. 4.2-6

(39) $\bar{\alpha}_{k+1}$ preserves the order of dependent accesses of α_{k+1}

                                (36)+(37)+(38)+Def. 6.3-1

(40) $\bar{\alpha}_{k+1}$ is equivalent to α_{k+1}; i.e., D for $\bar{\alpha}_{k+1}$ (35)+(39)+Thm. 6.3-2

Thus it is proven by induction that ω_n is equivalent to ω and $\bar{\alpha}_n$ is

equivalent to α_n.

(41) $\alpha_n = \omega'$ and $\beta_n = \omega - \bar{\alpha}_n = \lambda$ (2)+(1)

(42) $\omega_n = \bar{\alpha}_n \beta_n = \bar{\alpha}_n$ (2)+(41)

(43) ω_n is equivalent to ω' D+(41)+(42)

(44) ω is equivalent to ω' (43)+C

Since this is true for any two computations in any one job in J,

(45) (Int, J) is determinate   Def. 6.1-1

                                                              Q.E.D.

Chapter 7

Proof of the Functionality of L_D

The purpose of this chapter is to demonstrate that any L_D program running on the modified interpreter M is functional. In accordance with the plan presented at the beginning of Chapter 4, it will first be proven that every expansion in the corresponding entry-execution model $EE(L_D,M)$ is determinate. Then it will be shown that the expansion of a program P is determinate only if P is functional.

Chapter 6 has just concluded with a general result for any Structure-as-Storage (S-S) model: An expansion is determinate if it satisfies the Determinacy Axioms. Section 7.1 below verifies that $EE(L_D,M)$ is an S-S model. Section 7.2 then proves that every expansion in that model satisfies the Axioms, and so is determinate. Finally, Section 7.3 demonstrates that the construction of $EE(L_D,M)$ produces determinate expansions only from functional programs.

7.1 Verification that EE(L_D,M) is an S-S Model

An S-S model is defined fundamentally by a set of constraints on the computations in every job from an entry-execution model. These constraints were synthesized from the schema model of L_{BS} on the standard interpreter, so that EE(L_{BS},S) would satisfy them. Chapter 5 validates this construction by proving analytically that EE(L_{BS},S) is indeed an S-S model. L_D is a subset of L_{BS}, and the modifications to the standard interpreter did not change the actions performed by the structure operations. Therefore, it is to be expected not only that EE(L_D,M) is also an S-S model, but that the proof of this is very similar to that for EE(L_{BS},S). A brief review of the principle of the earlier proof will serve to motivate the steps taken in this section.

The first constraint is easily proven, and the second is a special case (handled here is Section 7.1.2). The remaining five constraints are verified by a three-step deduction: First, the constraints are satisfied by every canonical computation, or pair of canonical computations, as appropriate, in every job from EE(L_{BS},S). For any two computations α_1 and α_2 in a job of interest, there is a pair of these canonical computations ω_1 and ω_2 such that:

A: For i=1,2, for any execution e_1 initiated in α_i and any other execution e_2, e_1 is in reach $R(e_2)$ in α_i iff e_1 is in $R(e_2)$ in ω_i.

B: For any two pointers p_1 and p_2, $(p_1,\alpha_1)\rho(p_2,\alpha_2) = (p_1,\omega_1)\rho(p_2,\omega_2)$.

Finally, A and B imply that the five constraints, known to be satisfied by ω_1 and ω_2, must hold for α_1 and α_2.

The validation of $EE(L_D,M)$ as an S-S model is a similar deduction. The first and third steps are identical to those just listed, and so the proofs in Chapter 5 apply directly. The second step here requires an extension of the technique used earlier. A and B hold between any two pairs of computations α_1, α_2 and ω_1, ω_2 if either, for i=1,2, α_i is a prefix of ω_i, or for i=1,2, ω_i is SOE-inclusive of α_i (Lemmas 5.2-6 and 5.3-10). Accordingly, the following chain of computations is exhibited:

α – any computation in any job from $EE(L_D,M)$

β – a computation in $J_{S,\Omega}$, where S is an initial modified state and Ω is a halted firing sequence starting in S, which has α as a prefix

$\omega = \eta(S,\Omega)$ – a computation which is SOE-inclusive of β (Lemma 5.2-7)

$\omega' = \eta(S',\Omega')$, where S' is the initial standard state corresponding to S and Ω' is a halted firing sequence starting in S' and having Ω as a prefix – a canonical computation from $EE(L_{BS},S)$ which is SOE-inclusive of ω (to be proven)

For any pair α_1 and α_2, the corresponding ω'_1 and ω'_2 satisfy the final five constraints (Lemmas 5.3-3, 5.3-5, and 5.3-6). The general results mentioned are applied to each successive pair of computations in the chain to show that A and B are true of α_i and ω'_i. Then by Lemma 5.3-11, α_1 and α_2 satisfy the five constraints.

The key task remaining here is to prove that ω' is SOE-inclusive of ω. For any execution $e = Ex(d,k)$ which has input entries in ω, there is a prefix $\theta\varphi$ of Ω in which φ is the kth firing of d, and each such input entry describes the removal by φ of a token in $S \cdot \theta$. Since $\theta\varphi$ is also a

prefix of Ω', e will have input entries in ω' describing the removal of

tokens in $S'\cdot\theta$. Similarly, e's output entries in ω (ω') describe the

removal of tokens on d's output arcs in $S\cdot\theta\phi$ ($S'\cdot\theta\phi$). Therefore, the

relation between e's input (or output) entries in ω and ω' depends on

the relation between $S\cdot\theta$ and $S'\cdot\theta$ (or $S\cdot\theta\phi$ and $S'\cdot\theta\phi$), which is eluci-

dated in Section 7.1.1 below. Section 7.1.2 then presents the result

that, as on the standard interpreter, any two equal firing sequences

starting in equal modified states yield equal final states. This is of

immediate importance in the special case of proving that every job

satisfies the second (Pointer Transparency) constraint. It is also used

extensively in showing that every expansion satisfies the Determinacy

Axioms. Section 7.1.3 then proves that ω' is SOE-inclusive of ω; the

proof that $EE(L_{BS},S)$ is an S-S model is then easily extended to verify

that $EE(L_D,M)$ is an S-S model.

7.1.1 A Comparison of Standard and Modified States

The purpose here is compare the states of the standard and modified

interpreters when started in corresponding initial states and subjected

to the same sequence of firings. The form of a modified interpreter

state, as detailed in Section 3.3, differs from that of a standard state

in two regards: the replacement of simple pointers as values of tokens by

read and write pointers, and the presence of a pool component Q.

Paralleling these are the following differences in content between an

initial modified state S and its corresponding initial standard state S':

every token with pointer value p in S' is replaced with one with value

(p,R), and the pool component in S is empty.

Any firing sequence Ω starting in S is an abbreviation for a sequence of states beginning with S. Each state in this latter sequence is obtained from the immediately-preceding one by an application of the state-transition rule. Assuming that Ω is also a firing sequence starting in S', the differences in content between the resulting final states $S \cdot \Omega$ and $S' \cdot \Omega$ are determined by differences in the state-transition rules of the two interpreters. The standard rule was modified just in that the data tokens output by a firing of a Select actor may be withheld. Specifically, if a firing of a Select actor labelled S would output a pointer p on the standard interpreter, and there are tokens of value (p,W) in the current configuration, then no data output tokens appear, and S is placed in the pool $\dot{Q}(p)$. S remains in Q(p), and the data-output arcs remain empty, until all tokens of value (p,W) have disappeared. Then S is removed from Q(p) and tokens of value (p,R) are placed on the data-output arcs of the actor labelled S.

From this, it is expected that these differences in content between $S \cdot \Omega$ and $S' \cdot \Omega$ would be observed:

(1) Every arc which holds a token of value (p,R) or (p,W) in $S \cdot \Omega$ holds a token of value p in $S' \cdot \Omega$.

(2) For every label S in any pool Q(p) for a pointer p in $S \cdot \Omega$, the data-output arcs of the actor labelled S are empty.

Point (2) implies that the data-output arcs of the actor labelled S necessarily hold tokens of value p in $S' \cdot \Omega$, by the following reasoning: Ω can be written as $\theta \varphi \Delta$, where φ is the last firing of S in Ω. Since S is in Q(p), that firing would have output tokens of value p on the

standard interpreter; hence, there are such tokens in the standard state $S' \cdot \theta\varphi$. For any prefix X of Δ, the data-output arcs of S in $S \cdot \theta\varphi X$ remain empty, so there can be no firing in Δ which removes a token from one of those arcs on the modified interpreter. Therefore, there is no firing of an actor in Δ which removes a token from one of those arcs on the standard interpreter, so there are still tokens of value p on the data-output arcs of S in $S' \cdot \Omega$. Any standard and modified states of the same program which are so related are <u>congruent</u>, as defined in the following:

<u>Definition 7.1-1</u> (Congruent states) Given any L_{BS} program P, let $S = (\Gamma,U,Q)$ be any modified state for P and let $S' = (\Gamma',U')$ be any standard state for P. For any arc b in P, the <u>conditions of b in S and S' match to within withheld outputs</u> iff:

   a. if b is a data-output arc of a Select operator labelled S and there is a pointer p such that S is in Q(p), then b holds a token of value p in Γ' and is empty in Γ;

   b. otherwise, either

      i. b is empty in both Γ and Γ', or

      ii. b holds tokens of non-pointer value v in both Γ and Γ', or

      iii. b holds a token of pointer value p in Γ' and a token of value (p,R) or (p,W) in Γ.

S and S' are <u>congruent</u>, written $S'\mu S$, iff

   1. U and U' are identical, and

   2. for each arc b in P, the conditions of b in S and S' match to within withheld outputs.

$\triangle$

Now it can be shown that a given firing sequence takes any initial modified state and its corresponding initial standard state into congruent final states:

Theorem 7.1-1 Let S be any initial modified state for any L_{BS} program P and let S' be the corresponding initial standard state. Let Ω be any firing sequence starting in S on the modified interpreter. Then

A: Ω is also a firing sequence starting in S' on the standard interpreter, and

B: $S' \cdot \Omega \mu S \cdot \Omega$.

Proof: (The proof of this is simply an exhaustive demonstration that, except for the differences noted, every firing of an actor has the same *effects* — *on the heap, its input arcs, its output arcs, and all other arcs* — on the two interpreters; thus, it is relegated to Appendix E.)

$\triangle$

Corollary 7.1-1 Let S be any initial modified state for any L_{BS} program P, and let θ be any firing sequence starting in S. Then for any label d of a Select operator in P, if d is in a pool in $S \cdot \theta$, then all of the data-output arcs of that operator are empty in $S \cdot \theta$.

Proof: Let S' be the initial standard state corresponding to S. Then $S' \cdot \theta \mu S \cdot \theta$ [Thm. 7.1-1]. The Corollary follows from this and Def. 7.1-1.

$\triangle$

7.1.2 Pointer Transparency

The Pointer Transparency Constraint is the only non-trivial constraint which does not lend itself to a proof in the form discussed at the start of Section 7.1. In common with the others, however, the proof

that every job satisfies this constraint parallels the corresponding

portion of the validation of $EE(L_{BS},S)$. The heart of the latter is

Theorem 5.3-1: for any two equal standard states S_1 and S_2, and any two

equal firing sequences Ω_1 starting in S_1 and Ω_2 starting in S_2, $S_2 \cdot \Omega_2$

equals $S_1 \cdot \Omega_1$. Developing an analogous result for the modified interpreter

requires first a definition of equal modified states.

Two standard states S_1' and S_2' for program P are equal iff there is a

one-to-one mapping I such that, for every arc b in P,

$Match((b,S_1'), I, (b,S_2'))$. The first of the two ways in which the form

of a modified state differs from that of a standard state is the presence

of read and write pointers, which are distinct; i.e., $(p,R) \neq (p',W)$, even

if $p = p'$. An appropriate definition for matching conditions of an arc

in two modified states has already been presented as Definition 3.4-1.

The second difference in form is the pool component in a modified state.

Intuitively, the pool components Q_1 and Q_2 of two states S_1 and S_2 are

equal only if, for every Select label S, $\exists p_1 : S \in Q_1(p_1) \leftrightarrow \exists p_2 : S \in Q_2(p_2)$.

Furthermore, if S is in a pool in both Q_1 and Q_2, then the pointers p_1

and p_2 must be related: Eventually, S will be removed from those pools,

and tokens of value (p_1,R) and (p_2,R) will be placed on its data-output

arcs; in order that the conditions of those arcs match at that time, p_1

and p_2 should point to equal components of the heaps in S_1 and S_2. This

is made precise in the following complete specification of two equal

modified states:

<u>Definition 7.1-2</u> Two modified interpreter states $S_1 = (\Gamma_1, U_1, Q_1)$ and

$S_2 = (\Gamma_2, U_2, Q_2)$ for the same program P are <u>equal</u> iff there is a single

one-to-one mapping I such that:

1. For every arc b in P, Match((b,S_1), I, (b,S_2)).

2. For every label S of a Select operator in P, letting $U_1 = (N_1,\Pi_1,SM_1)$ and $U_2 = (N_2,\Pi_2,SM_2)$,

$$\exists p_1: S \in Q_1(p_1) \Leftrightarrow \exists p_2: S \in Q_2(p_2) \Rightarrow U_2 \cdot \Pi_2(p_2) \overset{I}{=} U_1 \cdot \Pi_1(p_1).$$

$\triangle$

The proof that equal firing sequences Ω_1 and Ω_2 starting in equal initial modified states S_1 and S_2 yield equal final states proceeds indirectly through standard states. That is, the initial standard states S_1' and S_2' corresponding to S_1 and S_2 are equal to each other, so $S_1' \cdot \Omega_1$ equals $S_2' \cdot \Omega_2$. Since $S_1' \cdot \Omega_1 \mu S_1 \cdot \Omega_1$ and $S_2' \cdot \Omega_2 \mu S_2 \cdot \Omega_2$, it is easily shown that the condition of any arc b which holds a token in $S_2 \cdot \Omega_2$ matches its condition in $S_1 \cdot \Omega_1$: If b holds a token of non-pointer value v in $S_2 \cdot \Omega_2$, then it does so in $S_2' \cdot \Omega_2$ (by congruence), in $S_1' \cdot \Omega_1$ (by equality of standard states), and in $S_1 \cdot \Omega_1$ (by congruence again). If b holds a token of value (p_2,R) or (p_2,W) in $S_2 \cdot \Omega_2$, then it holds a token of value p_2 in $S_2' \cdot \Omega_2$, one of value p_1, which points to an equal component, in $S_1' \cdot \Omega_1$, and one of value (p_1,R) or (p_1,W) in $S_1 \cdot \Omega_1$. All that remains to show a match is to prove that b holds either a write pointer in both states or a read pointer in both states.

Proof that the pool components are equal is by an induction based on the following: For each prefix $\theta_1\varphi_1$ of Ω_1, the last firing φ_1 causes a Select label S to be placed into a pool $Q_1(p_1)$ iff, for the same-length prefix $\theta_2\varphi_2$ of Ω_2, φ_2 causes S to be placed in a pool $Q_2(p_2)$. Pointers p_1 and p_2 are those which would be placed on the output arcs of the actor labelled S by φ_1 and φ_2 on the standard interpreter; since $S_2' \cdot \theta_2\varphi_2$ equals

$S_1^!\cdot\theta_1\varphi_1$, p_1 and p_2 point to equal components. Finally, every arc holds a pointer (p_1,W) after a prefix of Ω_1 iff it holds a pointer (p_2,W) after the *same-length prefix of* Ω_2, so S is removed from the pools after the same-length prefix of each firing sequence.

Theorem 7.1-2 For any two equal modified states S_1 and S_2 for the same L_{BS} program P, and for any two equal firing sequences Ω_1 starting in S_1 and Ω_2 starting in S_2, $S_2\cdot\Omega_2$ equals $S_1\cdot\Omega_1$. Furthermore, if I is the mapping under which the conditions of each arc b in P match in S_2 and S_1, then the mapping under which the conditions of b match in $S_2\cdot\Omega_2$ and $S_1\cdot\Omega_1$ is $I\cup\{(n_1,n_2)\mid \exists k:\text{for }i=1,2,\ n_i$ is the node in the kth Copy firing in $\Omega_i\}$. Finally, the initial standard states corresponding to S_1 and S_2 are equal.

Proof:

(1) For any modified configuration Γ, let $DT(\Gamma)$ denote the configuration obtained by replacing each token in Γ of value (p,R) or (p,W), where p is a pointer, with a token of value p. Let $S_x = (\Gamma_x,U_x,Q_x)$ and $S_y = (\Gamma_y,U_y,Q_y)$ be any two modified states for P which are equal under a mapping K. Then $(DT(\Gamma_x),U_x)$ and $(DT(\Gamma_y),U_y)$ are standard states   Defs. 3.3-4+3.3-3+2.1-3

(2) For each arc b in P, $\text{Match}((b,S_x),\ K,\ (b,S_y))$ (1)+Def. 7.1-2

(3) Letting $U_x = (N_x,\Pi_x,SM_x)$ and $U_y = (N_y,\Pi_y,SM_y)$, either b has no token in Γ_x and Γ_y, or b has tokens of non-pointer value v in both Γ_x and Γ_y, or there are pointers p_1 and p_2 such that b has tokens of values (p_1,R) and (p_2,R), or (p_1,W) and (p_2,W), in Γ_x and Γ_y, and $U_y.\Pi_y(p_2) \overset{K}{=} U_x.\Pi_x(p_1)$ (2)+Defs. 3.3-3+3.4-1

(4) Either b has no token in either $DT(\Gamma_x)$ or $DT(\Gamma_y)$, or b has tokens

of non-pointer value v in $DT(\Gamma_x)$ and $DT(\Gamma_y)$, or there are pointers

p_1 and p_2 such that b has tokens of value p_1 and p_2 in $DT(\Gamma_x)$ and

$DT(\Gamma_y)$ and $U_x \cdot \Pi_x(p_1) \overset{K}{\equiv} U_y \cdot \Pi_y(p_2)$ (1)+(3)

(5) For each arc b, $Match((b,(DT(\Gamma_x),U_x)), K, (b,(DT(\Gamma_y),U_y)))$

                          (1)+(4)+Def. 2.4-2

(6) $(DT(\Gamma_x),U_x)$ equals $(DT(\Gamma_y),U_y)$ under K   (5)+Def. 2.4-3

(7) For i=1,2, let $S_i = (\Gamma_i,U_i,Q_i)$, where $U_i = (N_i,\Pi_i,SM_i)$. Then the

initial standard state corresponding to S_i is $(DT(\Gamma_i),U_i)$, so

the initial standard states corresponding to S_1 and S_2 are equal

under I   (1)+(6)+Def. 3.3-5

Prove the rest of the Theorem by induction on the length of Ω_1.

Basis: $|\Omega_1| = 0$.

(8) $|\Omega_2| = 0$ Def. 2.4-5

(9) $S_2 \cdot \Omega_2 = S_2$ and $S_1 \cdot \Omega_1 = S_1$ (8)+Def. 2.3-1

(10) $I \cup \{(n_1,n_2) \mid \exists k: \text{for } i=1,2, n_i \text{ is the node in the } k^{th} \text{ Copy firing}$

$\text{in } \Omega_i\} = I$ (8)

The Theorem follows from (9) and (10) plus the hypothesis.

Induction step: Assume that the Theorem is true for any Ω_1 and Ω_2 of

length $n \geq 0$, and consider equal firing sequences $\Omega_1 \varphi_1$ and $\Omega_2 \varphi_2$ of

length n+1, in which φ_1 is a firing of actor d.

(11) φ_2 is also a firing of d, and Ω_1 and Ω_2 are equal firing sequences

of length n   Def. 2.4-5

(12) Ω_i is a firing sequence starting in S_i (11)+Def. 2.3-1

(13) Use the following notation, for i=1,2:

$\text{Fire}(S_1 \cdot \Omega_1, d) = (\Gamma_1', U_1', Q_1')$, where $U_1' = (N_1', \Pi_1', SM_1')$

$S_1 \cdot \Omega_1 \varphi_1 = (\Gamma_1'', U_1'', Q_1'')$, where $U_1'' = (N_1'', \Pi_1'', SM_1'')$

Denote by Γ_1^s the configuration $\text{Standard}_\Gamma((\text{Strip}(\Gamma_1, d), U_1), d)$

Denote by I^+ the map $I \cup \{(n_1, n_2) \mid \exists k: \text{ for } i=1,2, n_i \text{ is the node in}$

the kth Copy firing in $\Omega_1\}$ and by I^* the map $I \cup \{(n_1, n_2) \mid \exists k:$

for $i=1,2, n_i$ is the node in the kth Copy firing in $\Omega_1 \varphi_1\}$

(14) $S_2 \cdot \Omega_2$ equals $S_1 \cdot \Omega_1$ under I^+ (12)+(13)+ind. hyp.

(15) For $i=1,2$, let $S_i^- = (DT(\Gamma_i), U_i)$. Then S_i^- is a standard state, and

S_2^- equals S_1^- under I^+ (14)+(1)+(6)

(16) d is enabled in $S_2 \cdot \Omega_2$ and in $S_1 \cdot \Omega_1$ (11)+Def. 2.3-1

(17) The distribution of tokens on d's input and output arcs in Γ_i,

hence in $DT(\Gamma_i)$, conforms to the enabling conditions for d

       (16)+(1)+Def. 3.3-6

(18) d is enabled in S_2^- and in S_1^- (17)+Def. 2.1-4

(19) φ_2 and φ_1 are two equal, length-1 firing sequences starting in S_2^-

and S_1^- (11)+(18)+Defs. 2.4-5+2.3-1

(20) $S_2^- \cdot \varphi_2$ equals $S_1^- \cdot \varphi_1$ under $I^+ \cup \{(n_1, n_2) \mid \exists k: \text{ for } i=1,2, n_i \text{ is the node}$

in the kth Copy firing in $\varphi_1\} = I^*$ (15)+(19)+(13)+Thm. 5.3-1

(21) For any configuration Γ, heap U, and actor d, the values of the

tokens on d's output arcs in $\text{Standard}_\Gamma((\Gamma, U), d)$ depend only on

the values of the tokens on d's input arcs in Γ and on U, and, if

d is a Copy, on the arbitrary pointer-node pair added to Π

       Defs. 2.1-5+2.2-5

(22) $S_1^- \cdot \varphi_1 = (\text{Standard}_\Gamma((DT(\Gamma_i), U_i), d), \text{Standard}_U((DT(\Gamma_i), U_i), d))$

       (19)+(11)+(15)+Defs. 2.3-1+3.3-7

(23) $U_1' = \text{Standard}_U((\text{Strip}(\Gamma_i, d), U_i), d)$ (13)+Def. 3.3-9

(24) d is a Select ⇒ its input arcs hold tokens of the same value in

DT(Γ_1) and Strip(Γ_1,d)   (10+Def. 3.3-8

(25) ∧ Standard$_U$((DT(Γ_1),U_1),d) = U_1 Def. 2.2-5

(26) ⇒ U_1' = U_1 (23)+Def. 2.2-5

(27) d is a Select ∧ there is a pointer on a data-output arc of d in

either Γ_1^s or Γ_2^s ⇒ there is a pointer on a data-output arc of d in

either Standard$_\Gamma$((DT(Γ_1),U_1),d) or Standard$_\Gamma$((DT(Γ_2),U_2),d)

(13)+(21)+(24)

(28) ⇒ there are two pointers p_1 and p_2 such that there are tokens of

value p_i on d's data-output arcs in Standard$_\Gamma$((DT(Γ_i),U_i),d) and

$U_2.\Pi_2(p_2) \overset{I^*}{=} U_1.\Pi_1(p_1)$ (22)+(25)+(20)+Defs. 2.4-3+2.4-2

(29) ⇒ there are tokens of value p_i on d's data-output arcs in Γ_i^s and

$U_2'.\Pi_2'(p_2) \overset{I^*}{=} U_1'.\Pi_1'(p_1)$ (13)+(21)+(24)+(26)+Def. 2.4-1

(30) For any actor c, $\exists p_1$: c∈$Q_1(p_1)$ iff $\exists p_2$: c∈$Q_2(p_2)$ and if so,

$\Pi_2(p_2) = I^+(\Pi_1(p_1))$ (14)+Defs. 7.1-2+2.4-1

(31) For any actor c, $\exists p_1$: c∈$Q_1'(p_1)$ iff $\exists p_1$: c∈$Q_1(p_1)$ or c = d is a

Select ∧ $\exists p_1$: p_1 is on the data-output arcs of d in Γ_1^s Def. 3.3-9

(32) iff $\exists p_2$: c∈$Q_2(p_2)$ (30)

(33) or c = d is a Select ∧ $\exists p_2$: p_2 is on the data-output arcs of d in Γ_2^s

(27)+(29)

(34) iff $\exists p_2$: c∈$Q_2'(p_2)$ Def. 3.3-9

(35) There are pointers p_1 and p_2 such that c∈$Q_1'(p_1)$ ⇒ c∈$Q_1(p_1)$ or

c = d is a Select and there are tokens of value p_1 on data-output

arcs of d in Γ_1^s Def. 3.3-9

(36) ⇒ since Π_1' is a subset of Π_1', $\Pi_2'(p_2) = I^+(\Pi_1'(p_1))$

(30)+(23)+Def. 2.2-5

(37) $\vee \ U_2^! . \Pi_2^! (p_2) \overset{*}{\underset{=}{L}} U_1^! . \Pi_1^! (p_1)$ (27)+(29)

(38) $\Rightarrow \Pi_2^! (p_2) = I^+ (\Pi_1^! (p_1))$ or $\Pi_2^! (p_2) = I^* (\Pi_1^! (p_1))$ Def. 2.4-1

(39) $\Rightarrow$ since I^+ is a subset of I^*, $\Pi_2^! (p_2) = I^* (\Pi_1^! (p_1))$ (13)

(40) Let S_1^* and S_2^* be the <u>initial</u> standard states corresponding to S_1

and S_2. Then, for i=1,2, $\Omega_i \varphi_i$ is a firing sequence starting in S_i^*,

and $S_1^* \cdot \Omega_1 \varphi_1 \mu S_1 \cdot \Omega_1 \varphi_1$ Thm. 7.1-1

(41) $S_2^* \cdot \Omega_2 \varphi_2$ equals $S_1^* \cdot \Omega_1 \varphi_1$ under I^*, which is one-to-one

(7)+(40)+(13)+Thm. 5.3-1

(42) For any input arc b of d, b has a token in Γ_1^s iff d is a merge

gate whose control input in $Strip(\Gamma_1,d)$ is <u>false</u> (<u>true</u>), b is its

T (F) input arc, and b has the same token in $Strip(\Gamma_1,d)$

(13)+Defs. 3.3-7+2.1-5

(43) For any arc b, $\exists p$: b has a token of value (p,W) in $\Gamma_1^!$ $\Rightarrow$ $\exists p$: b

has a token of value (p,W) in Γ_1^s and [b is an output arc of d and

d is a Copy or Select $\Rightarrow$ b is a number-1 output arc of a Copy]

Def. 3.3-9

(44) $\Rightarrow$ [b is an output arc of d and d is a Select or Copy $\Rightarrow$ b is a

number-1 output arc of a Copy] and either [$\exists p$: b is neither an

input nor an output arc of d and b holds a token of value (p,W)

in $Strip(\Gamma_1,d)$] or [$\exists p$: b is an input arc of d and holds a token

of value (p,W) in Γ_1^s] or [$\exists p$: b holds a token of value (p,W) in

Γ_1^s and b is an output arc of d, so d is either a Copy, Select,

or pI operator] Defs. 2.1-5+2.2-5

(45) $\Rightarrow$ either

(45a) $\exists p$: b holds a token of value (p,W) in $Strip(\Gamma_1,d)$, and either b

is not an input or an output arc of d, or d is a merge gate whose

control input in $\mathrm{Strip}(\Gamma_1,d)$ is <u>false</u> (<u>true</u>) and b is d's T (F) input arc, or

(45b) b is a number-1 output arc of d and d is a Copy, or

(45c) $\exists p$: b is an output arc of d, d is a pI operator, and b holds a token of value (p,W) in Γ_1^s (42)

(46) (45a) $\Rightarrow$ since a gate is a pI actor, $\exists p$: b holds a token of value (p,W) in Γ_1, and either b is not an input or output arc of d, or d is a merge gate whose control input in Γ_1 is <u>false</u> (<u>true</u>) and b is its T (F) input arc   Def. 3.3-8

(47) $\Rightarrow \exists p'$: b holds a token of value (p',W) in Γ_2 and either b is not an input or output arc of d, or d is a merge gate whose control input in Γ_2 is <u>false</u> (<u>true</u>) and b is d's T (F) input arc

     (14)+Defs. 7.1-2+3.4-1

(48) $\Rightarrow \exists p'$: b holds a token of value (p',W) in $\mathrm{Strip}(\Gamma_2,d)$, and either b is not an input or output arc of d, or d is a merge gate whose control input in $\mathrm{Strip}(\Gamma_2,d)$ is <u>false</u> (<u>true</u>) and b is its T (F) input arc   Def. 3.3-8

(49) $\Rightarrow \exists p'$: b holds a token of value (p',W) in Γ_2^s, hence in Γ_2'

     (42)+Defs. 2.1-5+3.3-9

(50) (45b) $\Rightarrow \exists p'$: there is a token of value (p',W) on b in Γ_2' Def. 3.3-9

(51) (45c) $\Rightarrow \exists p$: there is an input arc of d which holds a token of value (p,W) in $\mathrm{Strip}(\Gamma_1,d)$, hence in Γ_1, and if d is a T- (F-)gate, its control input arc in $\mathrm{Strip}(\Gamma_1,d)$, hence in Γ_1, is <u>true</u> (<u>false</u>)

     Defs. 3.3-8+2.2-4+2.1-5

(52) $\Rightarrow \exists p'$: there is an input arc of d which holds a token of value (p',W) in Γ_2, hence in $\mathrm{Strip}(\Gamma_2,d)$, and if d is a T- (F-)gate, its

control input in Γ_2, hence in $\text{Strip}(\Gamma_2, d)$, is <u>true</u> (<u>false</u>)

(14)+Defs. 3.3-8+7.1-2+3.4-1

(53) $\Rightarrow \exists p'$: b holds a token of value (p', W) in Γ_2^s, hence in Γ_2'

Defs. 2.2-4+2.1-5+3.3-9

(54) For any arc b, $\exists p$: b holds a token of value (p, W) in $\Gamma_1' \Rightarrow \exists p'$: b

holds a token of value (p', W) in Γ_2'

(43)+(45)+(46)+(49)+(50)+(51)+(53)

By symmetry,

(55) For any arc b, $\exists p$: b holds a token of value (p, W) in $\Gamma_2' \Rightarrow \exists p'$: b

holds a token of value (p', W) in Γ_1' (43)-(54)

(56) For any arc b, $\exists p$: b holds a token of value (p, W) in Γ_1'' iff $\exists p$: b

holds a token of value (p, W) in Γ_1' (13)+Def. 3.3-9

(57) iff $\exists p'$: b holds a token of value (p', W) in Γ_2' (54)+(55)

(58) iff $\exists p'$: b holds a token of value (p', W) in Γ_2'' Def. 3.3-9

(59) The heap in $S_1^* \cdot \Omega_1 \varphi_1$ is $U_1'' = U_1'$ (13)+(40)+Defs. 7.1-1+3.3-9

(60) Letting $S_1^* \cdot \Omega_1 \varphi_1$ be (Γ_1^*, U_1''), for every arc b in P, either

    b is empty in both Γ_1^* and Γ_2^*, or

    b has tokens of non-pointer value v in Γ_1^* and Γ_2^*, or

    there are pointers p_1 and p_2 such that b has a token of value p_1

    in Γ_1^* and $U_2'' \cdot \Pi_2''(p_2) \overset{\Gamma^*}{=} U_1'' \cdot \Pi_1''(p_1)$ (59)+(41)+Defs. 2.4-3+2.4-2

(61) For every arc b in P, [b is a data-output arc of Select S $\Rightarrow$ there is

    no pointer p such that $\text{SEQ}_1''(p)$ or $\text{SEQ}_2''(p)$] $\Rightarrow$ either

    b is empty in both Γ_1'' and Γ_2'', or

    b has tokens of non-pointer value v in Γ_1'' and Γ_2'', or

    b has a token of value (p_1, R) or (p_1, W) in Γ_1'' and

$$U_2'' \cdot \Pi_2''(p_2) \overset{I^*}{=} U_1'' \cdot \Pi_1''(p_1) \qquad\qquad \text{(13)+(40)+Def. 7.1-1}$$

(62) $\Rightarrow$ [$\exists p_1$: b has a token of value (p_1, W) in Γ_1'' $\Rightarrow$ $\exists p_2$: b has a token

of value either (p_2, R) or (p_2, W) in Γ_2'' and $U_2'' \cdot \Pi_2''(p_2) \overset{I^*}{=} U_1'' \cdot \Pi_1''(p_1)$

$\Rightarrow$ b has as token of value (p_2, W) in Γ_2'' and $U_2'' \cdot \Pi_2''(p_2) \overset{I^*}{=} U_1'' \cdot \Pi_1''(p_1)$]

(56)+(58)

(63) $\wedge$ [$\exists p_1$: b has a token of value (p_1, R) in Γ_1'' $\Rightarrow$ $\not\exists p_2$: b has a token

of value (p_2, W) in Γ_2'' $\Rightarrow$ $\exists p_2$: b has a token of value (p_2, R) in Γ_2''

and $U_2'' \cdot \Pi_2''(p_2) \overset{I^*}{=} U_1'' \cdot \Pi_1''(p_1)$] (56)+(58)

(64) $\Rightarrow$ Match$((b, (\Gamma_2'', U_2'', Q_2'')), I^*, (b, (\Gamma_1'', U_1'', Q_1'')))$ Def. 3.4-1

(65) For any Select operator c, $\exists p_2$: $c \in Q_2''(p_2)$ $\Rightarrow$ $\exists p_2$: $c \in Q_2'(p_2)$ and there

is an arc a in p which holds a token of value (p_2, W) in Γ_2',

hence in Γ_2'' (13)+Def. 3.3-9

(66) $\Rightarrow$ $\exists p_1$: $c \in Q_1'(p_1)$ and $\Pi_2'(p_2) = I^*(\Pi_1'(p_1))$ (31)+(34)+(35)+(39)

(67) $\wedge$ a is not a data-output arc of a Select operator Def. 3.3-9

(68) $\Rightarrow$ there is a pointer p_3 such that a has a token of value (p_3, W)

in Γ_1'', hence in Γ_1', and $U_2'' \cdot \Pi_2''(p_2) \overset{I^*}{=} U_1'' \cdot \Pi_1''(p_3)$

(65)+(61)-(63)+Def. 3.3-9

(69) $\Rightarrow$ $\Pi_2'(p_2) = I^*(\Pi_1'(p_1))$ and $\Pi_2''(p_2) = I^*(\Pi_1''(p_3))$ (66)+Def. 2.4-1

(70) $\Rightarrow$ $\Pi_2'(p_2) = I^*(\Pi_1'(p_1))$ and $\Pi_2'(p_2) = I^*(\Pi_1'(p_3))$ (59)

(71) $\Rightarrow$ $p_1 = p_3$, since I^*, Π_1', and Π_2' are one-to-one (41)+Def. 2.2-1

(72) $\Rightarrow$ $c \in Q_1'(p_1)$ and there is an arc which has a token of value (p_1, W)

in Γ_1' (66)+(68)

(73) $\Rightarrow$ $c \in Q_1''(p_1)$ and $U_2'' \cdot \Pi_2''(p_2) \overset{I^*}{=} U_1'' \cdot \Pi_1''(p_1)$ (71)+(68)+Def. 3.3-9

By symmetry,

(74) $\exists p_1$: $c \in Q_1''(p_1)$ $\Rightarrow$ $\exists p_2$: $c \in Q_2''(p_2)$ and $U_2'' \cdot \Pi_2''(p_2) \overset{I^*}{=} U_1'' \cdot \Pi_1''(p_1)$

(75) For any arc b, b is a data-output arc of a Select operator S and

there is a p such that either $S\epsilon Q_1''(p)$ or $S\epsilon Q_2''(p)$ $\Rightarrow$ there are

pointers p_1 and p_2 such that $S\epsilon Q_1''(p_1)$ and $S\epsilon Q_2''(p_2)$ (65)+(73)+(74)

(76) $\Rightarrow$ b is empty in both Γ_1'' and Γ_2'' (40)+Def. 7.1-1

(77) For any arc b in P, $Match((b,S_2 \cdot \Omega_2 \varphi_2), I^*, (b,S_1 \cdot \Omega_1 \varphi_1))$

$$(13)+(60)+(64)+(75)+(76)+\text{Def. } 3.4-1$$

(78) $S_2 \cdot \Omega_2 \varphi_2$ equals $S_1 \cdot \Omega_1 \varphi_1$ under I^* (77)+(65)+(73)+(74)+Def. 7.1-2

$$\triangle$$

This fundamental result will have many important applications in
this chapter, the first being in the proof that every job from $EE(L_D,M)$
satisfies the Pointer Transparency Constraint. This is composed of one
Corollary and one Lemma, whose statements are identical to those developed
in the validation of $EE(L_{BS},S)$ and whose proofs are so similar that only
the differences are noted here.

Corollary 7.1-2 Let S_1 be any modified state for an L_{BS} program P, and
let Ω_1 be any firing sequence starting in S_1. Let S_2 be any modified
state equal to S_1, and let Ω_2 be any firing sequence equal to Ω_1. Then

A: Each actor in P is enabled in S_2 iff it is enabled in S_1.

B: If the multiset AP of the pointer-node pairs in the Copy firings
   in Ω_2 is consistent with the heap in S_2, then Ω_2 is a firing
   sequence starting in S_2, and Ω_2 is halted iff Ω_1 is halted.

Proof: Identical to the proof of Corollary 5.3-1 with the following
exceptions:

Lines (1) through (4) should read

(1) There is some one-to-one mapping I under which, for each arc b in P,

Match$((b,S_2)$, I, $(b,S_1))$, and, letting the pool component in S_i be

Q_i, i=1,2, for each actor d, $\exists p: d \in Q_1(p) \Leftrightarrow \exists p': d \in Q_2(p')$ Def. 7.1-2

(2) For each actor d in P, each input and output arc of d has a token in

S_2 iff it has a token in S_1 (1)+Def. 3.4-1

(3) Enabling conditions for d depend only on the presence or absence of

tokens on d's input and output arcs and on whether or not d is

in a pool   Defs. 3.3-6+2.1-4

(4) d is enabled in S_2 iff d is enabled in S_1 (3)+(2)+(1)

Line (15) is replaced by the two lines:

(15a) $\Rightarrow$ letting $S_2 \cdot \theta_2$ be (Γ,U), (p,n) cannot be added to Π in going from

U to $Standard_U((Strip(\Gamma,d),U),d)$ Def. 3.3-9

(15b) $\Rightarrow$ letting U = (N,Π,SM), $p \in dom\, \Pi$ or $n \in N$ Table 2.2-1

$\triangle$

__Lemma 7.1-1__ Let (Int,J) be any expansion from $EE(L_D,M)$. Then every
job $J \in J$ satisfies the Pointer Transparency Constraint.

__Proof:__ Identical to the proof of Lemma 5.3-2 with the following
exceptions:

Line (3) should read:

(3) (Int,J) is the expansion of some L_D program P, which is also an L'_{BS}

program, and J = J_E for some equivalence class E of initial

modified states for P   Defs. 3.3-12+4.3-1+4.3-2

The following substitutions are made:

     Thm. 7.1-2   for   Thm. 5.3-1

     Cor. 7.1-2   for   Cor. 5.3-1

     Def. 7.1-2   for   Def. 2.4-3

Def. 3.4-1   for   Def. 2.4-2

in the justifications for lines (7), (8), (16), (18), (19), (20), (31), (57), and (59).

For i=1,2, the phrase "token with value v_1" is replaced with "token with value v_1, (v_1,R), or (v_1,W)" in lines (6), (19), (34), (37), (40), and (41).

Line (13) is replaced with the two lines

(13a) ⇒ (p,n) could not be added to Π' in going from U' to

      Standard$_U$((Strip(Γ',d),U'),d)   Table 2.1-1

(13b) ⇒ (p,n) could not be added to Π' in going from $S_1 \cdot \theta$ to $S_1 \cdot \theta\varphi$

                                                          Def. 3.3-9

$\triangle$

7.1.3 Relation Between Canonical Computations in $EE(L_D,M)$ and $EE(L_{BS},S)$

As explained at the start of this section, the major new development used to validate $EE(L_D,M)$ as an S-S model is the following assertion: For any initial modified state S and halted firing sequence Ω starting in S, there is an initial standard state S' and halted firing sequence Ω' starting in S' such that $\eta(S',\Omega')$ is SOE-inclusive of $\eta(S,\Omega)$. The prime candidate for S' is the initial standard state corresponding to S, for that is as closely related to S as any standard state can be. It has already been shown that Ω is a firing sequence starting in S' and that $S' \cdot \Omega \mu S \cdot \Omega$. Unfortunately, Ω cannot always be used for Ω', because it may not be a halted firing sequence starting in S'. This occurs in the case that $S \cdot \Omega$ is <u>hung-up</u>; i.e., has a non-empty pool component.

7.1.3.1 Hang-Ups

An example of a hung-up modified state is shown in Figure 7.1-1.

$Q(p) = \{S\}$
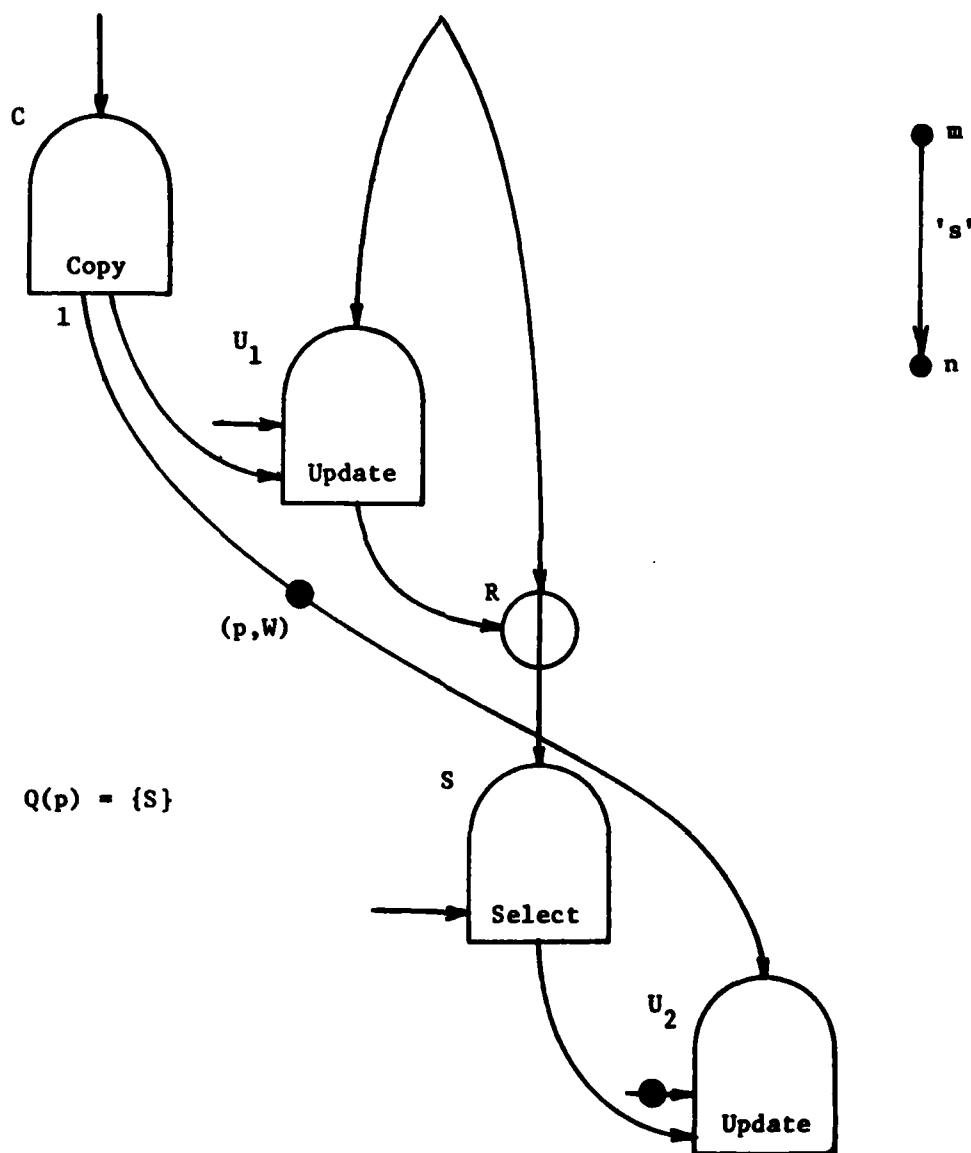
A Hung-Up Modified State

Figure 7.1-1

This depicts the result of the firing sequence $\Omega = (C,(p,n))$, U_1, R, S starting in some initial modified state S. The firing of C activated node n, the pointer to which is p, and left tokens of value (p,W) on C's output arcs. The firing of U_1 had as inputs q, the pointer to a second node m, selector 's', and p; its effect was to make n the 's'-successor of m. The firing of S also had q as its pointer input and its selector input happened to be 's'. On the standard interpreter, that Select firing would output a token of value p, enabling U_2; therefore, Ω starting in the corresponding initial standard state is not halted. On the modified interpreter, however, the label S is placed in Q(p) at the firing of the Select, and is not immediately removed because of the presence of the write pointer (p,W). Therefore, U_2 is not enabled in $S\cdot\Omega$, and Ω starting in S on the modified interpreter is not halted.

Figure 7.1-2 demonstrates why an unhalted Ω cannot be used as Ω' (i.e., why $\eta(S',\Omega)$ is not necessarily SOE-inclusive of $\eta(S,\Omega)$). It displays the same program as above with the addition of a First and another Select. A halted firing sequence for this program starting in any initial modified state S is $\Omega = (C,(p,n))$, U_1, R, S_1, F. As above, Ω is not halted when starting in the corresponding initial standard state S', because there will be tokens on both of S_1's output arcs, enabling U_2 and S_2. Since Ω is halted, the token left on F's number-1 output arc b in $S\cdot\Omega$ causes $\eta(S,\Omega)$ to have an entry whose transfer has source Source(b,S,Ω) = Src(Ex(F,1),1). Because Ω is not halted starting in S', however (and no firing in Ω removes a token from F's output arc), $\eta(S',\Omega)$ will have no entry whose transfer has that same source. I.e., e = Ex(F,1)
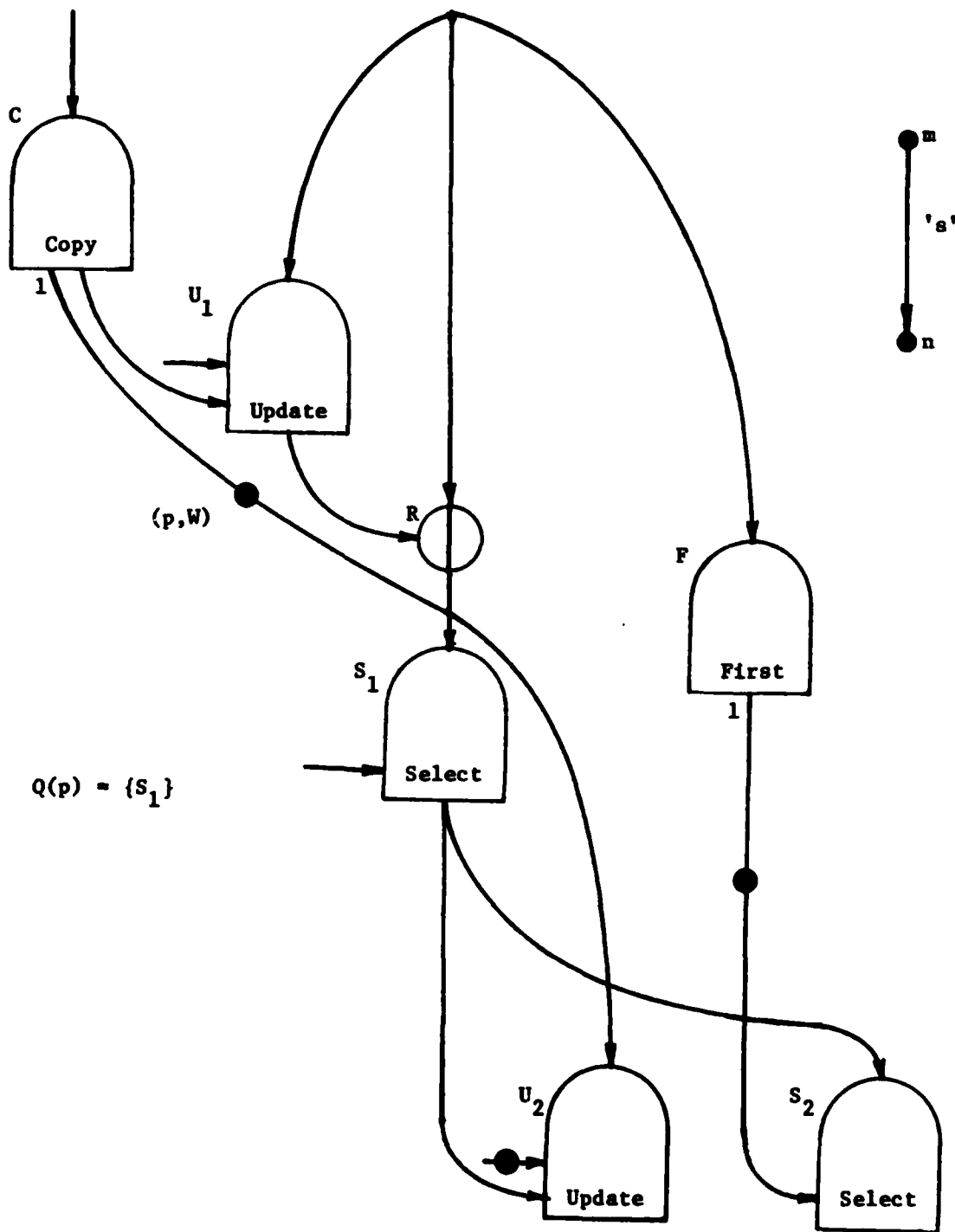
A Critical Hang-Up

Figure 7.1-2

has output entries in $\eta(S,\Omega)$ but not in $\eta(S',\Omega)$. Thus, Ω cannot be used as Ω', for at least the superficial reason that $\eta(S',\Omega)$ is not SOE-inclusive of $\eta(S,\Omega)$.

There is a deeper reason that Ω is unsuitable, the reason for including the requirement for inclusive sets of output entries in the definition of SOE-inclusive: The output entries of e in $\eta(S,\Omega)$ may be constrained to have a certain value. The technique being used to prove that they do relies on the existence of another, SOE-inclusive computation in which the output entries of e are known to have the constrained value. Then since the output entries of e in $\eta(S,\Omega)$ have the same value, they satisfy the constraint. Clearly this deduction would break down if there were no output entries of e in the SOE-inclusive computation.

Fortunately, the possibility of hang-ups on the modified interpreter does not invalidate any of the results of this thesis. In particular, for any halted firing sequence Ω_1 starting in any initial state S_1 of an L_D program, if $S_1 \cdot \Omega_1$ is hung-up, then for any other halted firing sequence Ω_2 starting in any state S_2 equal to S_1, $S_2 \cdot \Omega_2$ is a hung-up state which is equal to $S_1 \cdot \Omega_1$. I.e., L_D programs are functional on the modified interpreter, independent of the issue of hang-ups. Furthermore, the translation from L_{BV} programs to equivalent L_D programs (Algorithm 3.4-1) produces programs which do not hang up.

7.1.3.2 Discovering the SOE-Inclusive Computation

For each initial modified state S, corresponding initial standard state S', and halted firing sequence Ω starting in S, $\eta(S',\Omega)$ is not necessarily SOE-inclusive of $\eta(S,\Omega)$. This is because even though Ω is a

firing sequence starting in S', it might not be halted, if $S \cdot \Omega$ is a hung-up state. Ω is however a prefix of a halted firing sequence Ω' starting in S'. For any such Ω', $\omega' = \eta(S', \Omega')$ is SOE-inclusive of $\omega = \eta(S, \Omega)$ (Definition 5.2-8), as the following argument shows.

The computation $\eta(S', \Omega)$ is a prefix of ω'. It is apparent from Algorithm 4.3-1 that all structure operation executions initiated in ω are initiated, in the same order, in $\eta(S', \Omega)$, hence in ω'. For any non-pI execution $e = Ex(d,k)$, let $\theta\varphi$ be the prefix of Ω (and Ω') in which φ is the kth firing of d. Then there is an entry $Ent(e,j)$ of value v in ω $\Rightarrow$ there is a token of value v on d's number-j input arc in $S \cdot \theta$ which is removed by φ $\Rightarrow$ there is a token of value v on that arc in $S' \cdot \theta$ which is removed by φ (since $S' \cdot \theta \mu S \cdot \theta$) $\Rightarrow$ there is an entry $Ent(e,j)$ of value v in ω'. For any Copy execution $Ex(d,k)$ initiated in ω, there is a kth firing of operator d in Ω, so there is a prefix θ of Ω containing k firings of d such that there are tokens on d's output arcs in $S \cdot \theta$. Each such token keeps d disabled from firing a $k+1^{st}$ time until it is removed (if ever). Therefore, either it is removed by a subsequent firing in Ω which precedes the $k+1^{st}$ firing of d, or it is left in the final state $S \cdot \Omega$ and there are just k firings of d in Ω. In either case, there are output entries of $Ex(d,k)$ in ω (Lemma 7.1-2 below).

Finally, let f be any entry in ω, let $V(f)$ be v, and let the source in $T(f)$ be $Src(Ex(d,k),i)$. The target of f is an execution of an actor in P $\Rightarrow$ there is a prefix $\theta\varphi$ of Ω such that there is a token of value v on an output arc b of d in $S \cdot \theta$ which is removed by the immediately-following firing, and $Src(Ex(d,k),i) = Source(b,S,\theta)$ $\Rightarrow$ since $S' \cdot \theta \mu S \cdot \theta$, there is a token on b in $S' \cdot \theta$ which will be removed by the next firing, and since

both Source(b,S,θ) and Source(b,S',θ) depend primarily on the number of firings of d in θ, they are equal (Lemma 7.1-3 below) ⇒ there is an entry in ω' with value v whose transfer has the same source as T(f). The target of f is a dummy output execution ⇒ there is a token of value v on b in $S \cdot \Omega$ and there are k firings of d in Ω ⇒ since $S' \cdot \Omega \mu S \cdot \Omega$, there is a token of value v on b in $S' \cdot \Omega$ ⇒ d cannot fire a k+1st time in Ω' until that token is removed ⇒ either the token is removed by a subsequent firing in Ω' and k firings of d precede that removal, or the token is left on b in $S' \cdot \Omega'$ and there are k firings of d in Ω' ⇒ there is an entry in ω' with value v whose transfer has the same source as T(f).

The two lemmas cited above are proven first, each in a more general form which can be used in succeeding sections:

Lemma 7.1-2 Let S be any initial modified state for an L_{BS} program P, and let Ω be any halted firing sequence starting in S. There is a Copy firing (d,(p,n)) in Ω if and only if there are entries in η(S,Ω) with value p whose transfers have source Src(Ex(d,k),i) for some i, where (d,(p,n)) is the kth firing of d in Ω.

Proof: Prove "only if" first.

(1) Let Ξ be the prefix of Ω in which the last firing is (d,(p,n)). Then in $S \cdot \Xi$ there are tokens of value (p,R) or (p,W) on d's number-1 or number-2 output arcs. Let v be the value of one of those tokens   Defs. 2.1-2+2.3-1+3.3-9

(2) Let θ be the longest prefix of Ω such that for every prefix Λ of θ, $|\Xi| \leq |\Lambda|$, there is a token of value v on b in $S \cdot \Lambda$. Then d is not enabled in $S \cdot \Lambda$ Defs. 3.3-6+2.1-4

(3) Every firing of d in θ is in Ξ, so there are k firings of d in θ

(2)+Def. 2.3-1

(4) θ is not halted $\Rightarrow$ letting φ be such that $\theta\varphi$ is a prefix of Ω, φ is

not a firing of d   (2)+Def. 2.3-1+Cor. 7.1-1

(5) $\Rightarrow$ there is not a different token on b in $S\cdot\theta\varphi$ than in $S\cdot\theta$ Def. 3.3-9

(6) $\Rightarrow$ there is no token on b in $S\cdot\theta\varphi$; i.e., a token of value v was

removed in going from $S\cdot\theta$ to $S\cdot\theta\varphi$ (2)

(7) $\Rightarrow$ there is an entry in $\eta(S,\Omega)$ with value p whose transfer has

source Src(Ex(d,k),i) for some i   (1)+(3)+Alg. 4.3-1

(8) θ is halted $\Rightarrow$ $\theta = \Omega$ $\Rightarrow$ there is a token on b of value v in $S\cdot\Omega$ $\Rightarrow$

there is an entry in $\eta(S,\Omega)$ with value p whose transfer has source

Src(Ex(d,k),i) for some i   (1)+(2)+(3)+Alg. 4.3-1

Now prove "if".

(9) There is an entry in $\eta(S,\Omega)$ with value p whose transfer has source

Src(Ex(d,k),i) for some i, where d is a Copy operator $\Rightarrow$ there is a

prefix $\Delta\varphi$ of Ω containing exactly k firings of d such that tokens

of value (p,R) or (p,W) appear on d's number-i output arcs at the

transition from $S\cdot\Delta$ to $S\cdot\Delta\varphi$ Lemma 4.3-1

(10) $\Rightarrow$ φ is a firing of d which outputs tokens of value (p,R) or (p,W)

Def. 3.3-9

(11) $\Rightarrow$ φ is (d,(p,n)), and is the kth firing of d in Ω Def. 2.3-1

$\triangle$

Lemma 7.1-3 Given any L_{BS} program P, let b be any arc in P. Let S_1 and

S_2 be either any two equal initial modified states of P or one initial

modified state and the corresponding initial standard state. Let θ_1 and

θ_2 be any two firing sequences starting in S_1 and S_2 respectively such

that there is a token on b in both $S_1 \cdot \theta_1$ and $S_2 \cdot \theta_2$. If

(1) b is an output arc of an actor $\Rightarrow$ there are the same number of

firings of that actor in θ_1 and θ_2,

then Source(b, S_1, θ_1) = Source(b, S_2, θ_2). Furthermore, letting

Source(b, S_1, θ_1) be Src(Ex(c',n),i), if c' is in DL, then b is an output

arc of actor c $\Rightarrow$ there are zero firings of c in θ_1.

Proof:

(2) For any prefix $\Xi\varphi$ of θ_1 (θ_2), there is a token on b in $S_1 \cdot \Xi\varphi$

$(S_2 \cdot \Xi\varphi)$ which is not on b in $S_1 \cdot \Xi$ $(S_2 \cdot \Xi)$ $\Rightarrow$ b is an output arc of

an actor d and either φ is a firing of d or there is a pointer p

such that $d \in Q(p)$ in $S_1 \cdot \Xi$ $(S_2 \cdot \Xi)$ and b is a data-output arc of d

Defs. 3.3-9+3.3-7+2.1-5

(3) b is not an output arc of any actor $\Rightarrow \exists i$: b is the number-i program

input arc of P   Def. 2.1-1

(4) $\wedge$ there is no prefix $\Xi\varphi$ of θ_1 (θ_2) such that there is a token on b

in $S_1 \cdot \Xi\varphi$ $(S_2 \cdot \Xi\varphi)$ which is not on b in $S_1 \cdot \Xi$ $(S_2 \cdot \Xi)$ $\Rightarrow$ the token on

b in $S_1 \cdot \theta_1$ $(S_2 \cdot \theta_2)$ is on b in S_1 (S_2) (2)

(5) $\Rightarrow$ Source(b, S_1, θ_1) = Source(b, S_2, θ_2) = Src(Ex(ID,0),i) (3)+Alg. 4.3-1

(6) Assume b is an output arc of an actor d. Then b is not a program

input arc   Def. 2.1-1

(7) There is no token on b in S_1 (S_2) $\Rightarrow$ the token on b in $S_1 \cdot \theta_1$ $(S_2 \cdot \theta_2)$

is not on b in S_1 (S_2)

(8) There is a firing of d in θ_1 (θ_2) $\Rightarrow$ there is a prefix Ξ of θ_1 (θ_2)

such that d is enabled in $S_1 \cdot \Xi$ $(S_2 \cdot \Xi)$ Def. 2.3-1

(9) $\Rightarrow$ there is no token on b in $S_1 \cdot \Xi$ $(S_2 \cdot \Xi)$ $\Rightarrow$ the token on b in $S_1 \cdot \theta_1$

$(S_2 \cdot \theta_2)$ is not on b in S_1 (S_2)　　　　　　(6)+Defs. 3.3-6+2.1-4

(10) There is a token on b in S_1 (S_2) $\Rightarrow$ b is a control arc $\Rightarrow$ b is not

a data-output arc of a Select　　　　　　(6)+Defs. 3.3-5+2.2-6

(11) There is a token on b in S_1 (S_2) and there are zero firings of d in

θ_1 (θ_2) $\Rightarrow$ there is no prefix $\Xi\varphi$ of θ_1 (θ_2) such that the token

on b in $S_1 \cdot \Xi\varphi$ $(S_2 \cdot \Xi\varphi)$ is not on b in $S_1 \cdot \Xi$ $(S_2 \cdot \Xi)$ $\Rightarrow$ the token on

b in $S_1 \cdot \theta_1$ $(S_2 \cdot \theta_2)$ is on b in S_1 (S_2)　　　　　　(10)+(2)

(12) The token on b in $S_1 \cdot \theta_1$ $(S_2 \cdot \theta_2)$ is on b in S_1 (S_2) iff there is a

token on b in S_1 (S_2) and there are zero firings of c in θ_1 (θ_2)

　　　　　　(7)+(8)+(9)+(11)

(13) The token on b in $S_1 \cdot \theta_1$ is on b in S_1 iff there is a token on b

in S_1 and there are zero firings of d in θ_1　　　　　　(12)

(14) iff there is a token on b in S_2　　　Defs. 7.1-2+3.4-1+3.3-5

(15) and there are zero firings of d in θ_2　　　　　　(6)+(1)

(16) iff the token on b in $S_2 \cdot \theta_2$ is on b in S_2　　　　　　(12)

(17) The token on b in $S_1 \cdot \theta_1$ is on b in S_1 $\Rightarrow$ Source(b,S_1,θ_1) is

Src(Ex(IT,0),1) or Src(Ex(IF,0),1), according as the value of

that token is _true_ or _false_　　　　　　(10)+Alg. 4.3-1

(18) $\wedge$ the token on b in $S_2 \cdot \theta_2$ is on b in S_2　　　　　　(13)+(16)

(19) $\Rightarrow$ Source(b,S_2,θ_2) = Src(Ex(IT,0),1) or Src(Ex(IF,0),1) according

as the value of that token is _true_ or _false_　　　(10)+Alg. 4.3-1

(20) $\Rightarrow$ Source(b,S_2,θ_2) = Source(b,S_1,θ_1)　　　Defs. 7.1-2+3.4-1+3.3-5

(21) The token on b in $S_1 \cdot \theta_1$ is not on b in S_1 $\Rightarrow$ Source(b,S_1,θ_1) is

Src(Ex(d,k_1),i) where k_1 is the number of firings of d in θ_1 and

b is in the number-i group of output arcs of d　　　Alg. 3.4-1

(22) $\wedge$ the token on b in $S_2 \cdot \theta_2$ is not on b in S_2　　　　　　(13)+(16)

(23) $\Rightarrow$ Source(b,S_2,θ_2) = Src(Ex(d,k_2),i), where k_2 is the number of

firings of d in θ_2 Alg. 3.4-1

(24) $\Rightarrow$ Source(b,S_2,θ_2) = Source(b,S_1,θ_1)   (6)+(1)

(25) Source(b,S_2,θ_2) = Source(b,S_1,θ_1)   (3)+(5)+(6)+(17)+(20)+(21)+(24)

(26) Letting Source(b,S_1,θ_1) be Src(Ex(c',n),i), c'$\in$DL $\Rightarrow$ the token on b

in $S_1 \cdot \theta_1$ is on b in S_1 Def. 4.3-1+Alg. 3.4-1

(27) $\Rightarrow$ if b is an output arc of actor c, then there are zero firings

of c in θ_1 (6)+(12)

$\triangle$

Theorem 7.1-3 Let P be any L_{BS} program. For any initial modified state

S for P, let S' be the corresponding initial standard state, and let Ω

be any halted firing sequence starting in S. Then there is a halted

firing sequence Ω' starting in S' which has Ω as a prefix such that

$\eta(S',\Omega')$ is SOE-inclusive of $\eta(S,\Omega)$.

Proof: (The essence of the proof has already been expressed; the details

may be found in Appendix E.)

$\triangle$

As outlined at the start of this section, there is only a small

difference between the technique for validating EE(L_D,M) as an S-S model

and that for EE(L_{BS},S): the chain from any pair of computations α_1 and α_2

in a job to a pair of computations ω_1 and ω_2 known to satisfy the last

five constraints has one more link. Consequently, the proof of the

Theorem below is so similar to that for Theorem 5.3-1 that it has been

removed to Appendix E.

Theorem 7.1-4 EE(L_D,M) is a Structure-as-Storage model.

                                                Q.E.D.

7.2 Verification That EE(L$_D$,M) Satisfies the Determinacy Axioms

This section presents the proofs that any expansion (Int,J) from
EE(L$_D$,M) satisfies the seven Determinacy Axioms presented in Section 6.2.
Section 7.2.1 covers the first four axioms; each succeeding subsection
treats one of the remaining axioms: in order of increasing difficulty,
freedom from conflict, commutativity, and persistence. It is assumed
throughout that P is the L$_D$ program of which (Int,J) is the expansion
and that Int = (St, I,IE).

7.2.1 The First Four Axioms

The demonstration that (Int,J) satisfies these Axioms is simple:
For any job J$\in J$, each computation in J is causal and J has the Prefix
Property by construction. All actions except the eight structure oper-
ations are deterministic because all actors in an L$_D$ program except
structure operators have functions associated with them. For any e$\in$IE,
there is an arc b in P such that the values of the output entries of e in
any two computations in a job equal the values of the tokens on b in two
equal initial states of P; if either value is not a pointer, then they
are the same.

<u>Lemma 7.2-1</u> Every expansion (Int,J) from EE(L$_D$,M) satisfies the first
four Determinacy Axioms.

<u>Proof</u>:

(1) There is an L$_D$ program P of which (Int,J) is an expansion Def. 4.3-1

(2) Let α be any computation in any job J$\in J$. Then there is an initial
modified state S of P and a halted firing sequence Ω starting in
S such that α is a prefix of some $\beta\in J_{S,\Omega}$, and β itself is in J

(1)+Def. 4.3-3

(3) β is a causal permutation of $\eta(S,\Omega)$ (2)+Def. 4.3-5

(4) J is a job for Int, so β and α are computations for Int

Defs. 4.2-2+4.2-3

(5) Let γf be any prefix of α. Then γf is a prefix of β (2)

(6) Let e be the execution of which f is an output entry. Then e is

initiated in γ with respect to Int   (3)+(4)+(5)+Def. 4.2-7

(7) α is causal with respect to Int   (5)+(6)+(4)+Def. 4.2-7

(8) All prefixes of β are in J   (2)+Def. 4.3-3

(9) All prefixes of α are prefixes of β (2)

(10) J has the Prefix Property   (9)+(8)+Def. 4.2-7

(11) Let α_1 and α_2 be any two (not necessarily distinct) computations in

any two jobs J_1 and J_2 in J. Then there are two initial states

S_1 and S_2 for P and two halted firing sequences Ω_1 and Ω_2

starting in S_1 and S_2 such that, for i=1,2, α_i is a prefix of a

permutation β_i of $\omega_i = \eta(S_i,\Omega_i)$ Defs. 4.3-3+4.3-5

(12) Let Int = (St, I,IE). Let $e_1 = Ex(d_1,k_1)$ and $e_2 = Ex(d_2,k_2)$ be

any two executions not in IE such that $I(d_1) = I(d_2) = a$ is not a

structure operation and, for i=1,2, e_i has output entries in α_i.

Then e_i is initiated in α_i wrt Int; i.e., there are In(a) input

entries to e_i in α_i (7)+(4)+Def. 4.2-7

(13) Since e_i has output entries in α_i, it has output entries in ω_i, so

d_i is the label of an actor in P   (12)+Alg. 4.3-1

(14) Since that actor is not a structure operator, the only transitions

at which tokens appear on its output arcs are those caused by

firing it   (13)+Def. 3.3-9

(15) a is a function; i.e., the values of the tokens placed on d_i's

output arcs at any firing depend only on the values of the tokens

removed from d_i's input arcs at that firing

$$(12)+(13)+(1)+\text{Defs. } 3.3-12+2.2-3+2.1-2$$

(16) There are at most In(a) input entries to e_i in β_i, so β_i and hence

ω_i have the same set of input entries to e_i as does α_i

$$(11)+(4)+(12)+\text{Def. } 4.2-6$$

(17) For all j, there is an entry $\text{Ent}(e_1, j)$ in α_1 iff there is an entry

$\text{Ent}(e_2, j)$ in α_2, and if so, those entries' values are equal $\Rightarrow$

for all j, there is an entry $\text{Ent}(e_1, j)$ in ω_1 iff there is an

entry $\text{Ent}(e_2, j)$ in ω_2, and if so, their values are equal   (16)

(18) $\Rightarrow$ the k_1^{th} firing of d_1 in ω_1 and the k_2^{th} firing of d_2 in ω_2

remove tokens of the same values from the same set of input arcs

$$(13)+\text{Alg. } 4.3-1$$

(19) $\Rightarrow$ those firings place tokens of the same value on the output arcs

of d_1 and d_2 (15)

(20) $\Rightarrow$ for any i, the value of $\text{Src}(e_1, i)$ in ω_1, if any, equals the value

of $\text{Src}(e_2, i)$ in ω_2, if any $(14)+\text{Lemma } 4.3-1+\text{Def. } 4.2-6$

(21) $\Rightarrow$ for any i, the value of $\text{Src}(e_1, i)$ in α_1, if any, equals the value

of $\text{Src}(e_2, i)$ in α_2, if any   (11)

(22) All actions except the structure operations are deterministic

$$(11)+(12)+(17)+(21)+\text{Def. } 6.2-1$$

(23) Let α_1 and α_2 be any two computations in the same job $J \in J$. Then

there is an equivalence class E of initial modified states for P

such that $J = J_E$ Def. 4.3-2

(24) There are two equal initial modified states for P, S_1 and S_2, and

two halted firing sequences Ω_1 and Ω_2 starting in S_1 and S_2, such

that, for $j=1,2$, α_j is a prefix of a permutation of $\omega_j = \eta(S_j, \Omega_j)$

(23)+Defs. 4.3-3+4.3-5

(25) Let e be any execution in IE. For any i, the value of $Src(e,i)$ in

α_j equals the value of $Src(e,i)$ in ω_j (24)+Def. 4.2-6

(26) e is either $Ex(ID,0)$, $Ex(IT,0)$, or $Ex(IF,0)$ (25)+Def. 4.3-2

(27) $e = Ex(IT,0)$ ($e = Ex(IF,0)$) $\rightarrow$ the value of $Src(e,i)$ in ω_1 and ω_2 is

<u>true</u> (<u>false</u>)   (24)+Alg. 4.3-1

(27) $e = Ex(ID,0)$ $\rightarrow$ the value of $Src(e,i)$ in ω_1 (ω_2) is the value of

the token on the number-i program input arc of P in S_1 (S_2)

(24)+Alg. 4.3-1

(29) $\rightarrow$ the value of $Src(e,i)$ in ω_1 is not a pointer iff the value of

$Src(e,i)$ in ω_2 is not a pointer, and if the values are not

pointers, they are equal   (24)+Defs. 7.1-2+3.4-1

(30) The value of $Src(e,i)$ in α_1 is not a pointer iff the value of

$Src(e,i)$ in α_2 is not a pointer, and if those values are not

pointers, they are equal   (26)+(27)+(28)+(29)+(25)

(31) (Int, J) satisfies the first four Determinacy Axioms

(2)+(7)+(10)+(22)+(23)+(30)+Axioms 6.2-1-6.2-4

$\triangle$

7.2.2 Freedom From Conflict

This axiom concerns every two computations αgf and $\alpha \bar{f} \bar{g}$ in a job in

which $T(\bar{f}) = T(f)$, $T(\bar{g}) = T(g)$, and g and f initiate distinct executions

$e_2 = Ex(d_2,k_2)$ and $e_1 = Ex(d_1,k_1)$, respectively. It asserts that it is

not the case that in αgf, $Ent(e_1,1)$ and $Ent(e_2,1)$ are in the same access

history and e_1 is in the reach $R(e_2)$. There are two equal initial modified states S and S' for P, and two halted firing sequences Ω starting in S and Ω' starting in S', such that αgf is a prefix of some $\beta \in J_{S,\Omega}$ and $\alpha \overline{fg}$ is a prefix of some $\beta' \in J_{S',\Omega'}$. Since $\Phi(\beta)$ ($\Phi(\beta')$) is the reduction of Ω (Ω'), $\Phi(\alpha gf)$ ($\Phi(\alpha \overline{fg})$) is the reduction of a prefix of Ω (Ω') (Lemma 7.2-2 below). Since, for $i=1,2$, k_i serves as an index of executions of d_i (Corollary 4.3-1), e_i is the k_i^{th} execution of d_i to initiate in each of αgf and $\alpha \overline{fg}$. Therefore, the prefix of Ω (Ω') whose reduction is $\Phi(\alpha gf)$ ($\Phi(\alpha \overline{fg})$) is $\theta \varphi_2 \varphi_1$ ($\theta' \varphi_1' \varphi_2'$), in which, for $i=1,2$, φ_i (φ_i') is the k_i^{th} firing of d_i; furthermore, θ and θ' have the same reduction $\Phi(\alpha)$, so they are equal firing sequences.

Ent(e_1,1) and Ent(e_2,1) are in the same access history in αgf iff they have the same value iff φ_1 and φ_2 have equal number-1 pointer inputs. Given that, e_1 is in $R(e_2)$ iff the actions $I(d_1)$ and $I(d_2)$ are one of a certain few combinations, and, possibly, e_1 and e_2 have equal selector inputs. This in turn is iff φ_1 and φ_2 are firings of actors of that same combination of actions, and, possibly, have the same selector inputs in $\theta \varphi_2 \varphi_1$. Comparing the definitions of reach with Table 3.1-1, Ent(e_1,1) and Ent(e_2,1) are in the same access history and $e_1 \in R(e_2)$ iff φ_1 and φ_2 potentially interfere in $\theta \varphi_2 \varphi_1$.

The Commutativity Axiom also concerns two computations αgf and $\alpha \overline{fg}$ in the same job in which g and f may initiate distinct executions. The above result, therefore, is pertinent to both axioms, and so is stated separately below as Lemma 7.2-3; the essence of its proof has been conveyed well enough above that the details are deferred to Appendix E.

Lemma 7.2-2 Let S be any initial interpreter state and let Ω be any halted firing sequence starting in S. Let α be any prefix of any computation β in $J_{S,\Omega}$, and let θ be the prefix of Ω whose length equals the length of $\Phi(\alpha)$. Then the reduction of θ is $\Phi(\alpha)$.

<u>Proof</u>:

(1) $\Phi(\beta)$ is the reduction of Ω Def. 4.3-5

(2) A prefix of the reduction of Ω is the reduction of a prefix of Ω

                                                   Def. 2.4-5

(3) $\Phi(\alpha)$ is a prefix of $\Phi(\beta)$ Def. 4.3-4

(4) $\Phi(\alpha)$ is the reduction of a prefix Δ of Ω (3)+(2)+(1)

(5) The length of Δ equals the length of the reduction of Δ Def. 2.4-5

(6) $\Phi(\alpha)$ is the reduction of that prefix of Ω whose length is the same

      as the length of $\Phi(\alpha)$, i.e., θ (4)+(5)

$\triangle$

Lemma 7.2-3 For any equivalence class E of initial modified states for an L_{BS} program P, let J be J_E. Let Int(P) be (St, l, IE). Assume there are two computations $\alpha g f$ and $\alpha \overline{f} \overline{g}$ in J such that $T(\overline{f}) = T(f)$, $T(\overline{g}) = T(g)$, and f and g initiate distinct executions $e_1 = Ex(d_1, k_1)$ and $e_2 = Ex(d_2, k_2)$ in $\alpha g f$, where d_1 and d_2 are in St-DL. Let S and Ω (S' and Ω') be the state in E and halted firing sequence starting in that state such that $\alpha g f$ ($\alpha \overline{f} \overline{g}$) is a prefix of a computation in $J_{S,\Omega}$ ($J_{S',\Omega'}$). Then there are prefixes $\theta \varphi_2 \varphi_1$ of Ω and $\theta' \varphi_1' \varphi_2'$ of Ω', whose reductions are $\Phi(\alpha g f)$ and $\Phi(\alpha \overline{f} \overline{g})$, such that θ' equals θ and for i=1,2, φ_i (φ_i') is the k_i^{th} firing of d_i. Furthermore, φ_1 and φ_2 potentially interfere in $\theta \varphi_2 \varphi_1$ iff $Ent(e_1, 1)$ and $Ent(e_2, 1)$ are in the same access history, and e_1 is in $R(e_2)$, in $\alpha g f$.

$\triangle$

The proof that each expansion in $EE(L_D,M)$ satisfies the freedom-from-conflict axiom is by contradiction: By Lemma 7.2-3, if the axiom is not satisfied, there are two equal initial states S and S' and two firing sequences $\theta\varphi_2\varphi_1$ starting in S and $\theta'\varphi_1'\varphi_2'$ starting in S' in which, for $i=1,2$, φ_i (φ_i') is the $k_i{}^{th}$ firing of d_i; furthermore, S' equals S and φ_1 and φ_2 potentially interfere in $\theta\varphi_2\varphi_1$. By the Determinacy Condition, φ_1 and φ_2 are not in the same blocking group in $\theta\varphi_2\varphi_1$. It is argued at length in Section 3.2 that any two potentially-interfering firings in distinct blocking groups in any firing sequence starting in S are sequenced by S. I.e., the $k_1{}^{th}$ firing of d_1 must follow the $k_2{}^{th}$ firing of d_2 in all firing sequences starting in any state equal to S. There-fore, $\theta'\varphi_1'\varphi_2'$ cannot be a firing sequence starting in S'; hence a contra-diction.

This informal argument is presented rigorously in the following:

<u>Theorem 7.2-1</u> Every expansion (Int,J) from $EE(L_D,M)$ satisfies the Freedom-from-conflict Axiom.

<u>Proof</u>: By contradiction.

(1) Let P be the L_D program of which (Int,J) is an expansion. Assume that the Axiom does not hold for (Int,J)

(2) There is a computation αgf in a job $J\in J$ such that

(2a) f and g initiate distinct executions e_1 and e_2 respectively in αgf,

(2b) $Ent(e_1,1)$ and $Ent(e_2,1)$ are in the same access history in αgf,

(2c) e_1 is in the reach $R(e_2)$ in αgf, and

(2d) there is a computation $\alpha\bar{f}\bar{g}$ in J with $T(\bar{f}) = T(f)$ and $T(\bar{g}) = T(g)$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ (1)+Axiom 6.2-7

(3) There is an equivalence class E of initial modified states for P

such that $J = J_E$ Defs. 4.3-1+4.3-2

(4) There is an initial modified state S in E and a halted firing

sequence Ω starting in S such that $\alpha g f$ is a prefix of some β

in $J_{S,\Omega}$ (2)+(3)+Def. 4.3-3

(5) There is an initial modified state S' in E and halted firing

sequence Ω' starting in S' such that $\alpha \overline{\overline{f g}}$ is a prefix of some β'

in $J_{S',\Omega'}$ (2d)+(3)+Def. 4.3-3

(6) J is a job for Int = Int(P) = (St, l,IE)   (2)+Defs. 4.2-2+4.3-2

(7) $\alpha g f$ and $\alpha \overline{\overline{f g}}$ are both computations for Int(P)   (2)+(6)+Def. 4.2-3

(8) For i=1,2, let e_1 = Ex(d_1,k_1). Then $l(d_1)$ and $l(d_2)$ are both

structure operations, and the latter is an Assign, Update, or

Delete   ·   (2c)+(7)+Defs. 5.1-6+5.1-8

(9) d_1 and d_2 are both in St-DL   (7)+(8)+Def. 4.3-2

(10) There are prefixes $\theta\varphi_2\varphi_1$ of Ω and $\theta'\varphi_1'\varphi_2'$ of Ω', whose reductions

are $\Phi(\alpha g f)$ and $\Phi(\alpha\overline{\overline{fg}})$, such that θ' equals θ and, for i=1,2, φ_1

(φ_1') is the k_1^{th} firing of d_1. Furthermore, φ_1 and φ_2 potentially

interfere in $\theta\varphi_2\varphi_1$ iff Ent(e_1,1) and Ent(e_2,1) are in the same

access history, and e_1 is in R(e_2), in $\alpha g f$

                         (3)+(7)+(2)+(2d)+(2a)+(8)+(9)+(4)+(5)+Lemma 7.2-3

(11) φ_1 and φ_2 potentially interfere in $\theta\varphi_2\varphi_1$ (10)+(2b)+(2c)

(12) P satisfies the Determinacy Condition   (1)+Def. 3.3-12

(13) If φ_1 and φ_2 are in the same blocking group in Ω, then in any firing

sequence Ω' starting in any state S' equal to S, the k_1^{th} firing

of d_1 follows the k_2^{th} firing of d_2 (12)+(11)+(10)+Def. 3.3-11

(14) There is an S' equal to S and an Ω' starting in S' in which the k_1^{th}

firing of d_1 precedes the $k_2{}^{th}$ firing of d_2 (5)+(4)+(10)

(15) φ_1 and φ_2 are in distinct blocking groups in Ω (13)+(14)

(16) The actor labelled d_2 is a write-class operator(8)+Defs. 4.3-2+3.1-2

(17) P satisfies the Read-Only Condition   (1)+Def. 3.3-12

(18) d_2 is in the m.p.d.g. G(K) only if K = K(C,1) for some Copy operator

C   (16)+(17)+Def. 3.3-2

(19) P is an L_{BS} program   (1)+Def. 3.3-12

(20) P satisfies the Static/Dynamic Group Relationship   (19)+Lemma 3.3-1

(21) Exactly one of the following statements is true:

(21a) There is exactly one integer i such that $\varphi_2 \epsilon SB_\Omega(ID,i)$

(21b) There is exactly one Select operator S, one integer n and one

integer i such that $\varphi_2 \epsilon SB_\Omega(S,n,i)$

(21c) There is exactly one Copy operator C and one integer n such that

$\varphi_2 \epsilon SB_\Omega(C,n,2)$

(21d) There is exactly one Copy operator C and one integer n such that

$\varphi_2 \epsilon SB_\Omega(C,n,1)$

Furthermore, each of (21a), (21b), and (21c) $\Rightarrow$ dϵG(K) where K is

not K(C,1) for a Copy operator C   (4)+(10)+(20)+Def. 3.3-13

(22) There is exactly one Copy operator C and one integer n such that

$\varphi_2 \epsilon SB_\Omega(C,n,1)$ (21)+(18)

(23) The nth token to appear on C's number-1 output arcs has the same

value as the token removed from d_2's primary input arc by φ_2

    (22)+(20)+Def. 3.3-13

(24) For any Copy operator C, there is no token on C's output arcs in S

    Defs. 3.3-5+2.2-6

(25) A transition from empty to full condition for an output arc of C

occurs just at every firing of C   Defs. 3.3-6+2.1-4+3.3-9

(26) The nth token to appear on an output arc of C is output by the nth

firing of C   (24)+(25)

(27) There is a pointer p such that the value of the token removed from

d_2's primary input arc by φ_2, as well as the tokens placed on C's

number-1 group of output arcs at its nth firing, is (p,W)

(23)+(26)+Def. 3.3-9

(28) φ_1 has the same primary input as φ_2 (11)+Defs. 3.1-2+3.2-1

(29) φ_1 is not in $SB_\Omega(C,n,1)$ or $SB_\Omega(C,n,2)$ (22)+(15)+Def. 3.3-10

(30) There is a Copy operator C' and integer n' such that $\varphi_1 \in SB_\Omega(C',n',1)$

or $\varphi_1 \in SB_\Omega(C',n',2) \Rightarrow C' \neq C \vee n' \neq n$ (29)

(31) $\wedge$ the value of the token removed by φ_1 from d_1's primary input arc

equals the value of the n'^{th} token to appear on the number-1

or number-2 output arcs of C'   (20)+Def. 3.3-13

(32) $\Rightarrow$ the n'^{th} firing of C' places tokens of value (p,R) or (p,W) on the

output arcs of C'   (28)+(27)+(26)+Def. 3.3-9

(33) $\Rightarrow$ letting $\Delta\varphi_C$ be the prefix of Ω in which φ_C is the later of the

n^{th} firing of C and the n'^{th} firing of C', a Copy firing in Δ

outputs the same pointer p as φ_C (27)

(34) $\Rightarrow p \in dom \, \Pi$ in $S \cdot \Delta$ (30)+(4)+Lemma 5.2-1+Def. 2.3-1

(35) $\Rightarrow \varphi_C$ could not output (p,R) or (p,W)   Defs. 3.3-9+2.2-5

(36) For any Copy operator C' and integer n', $\varphi_1 \notin SB_\Omega(C',n',1)$,

$\varphi_1 \notin SB_\Omega(C',n',2)$, and [C' $\neq$ C $\vee$ n' $\neq$ n $\Rightarrow$ p is not output by the

n'^{th} firing of C']   (30)+(32)+(35)

(37) $\exists i$: $\varphi_1 \epsilon SB_\Omega(ID,i)$ $\Rightarrow$ there is a token of value (p,R) or (p,W) on a

program input arc in S (28)+(27)+(20)+Def. 3.3-13

(38) $\Rightarrow$ pϵdom Π in S Defs. 3.3-5+2.2-6

(39) Since there is a prefix $\Delta\varphi_C$ of Ω in which φ_C = (C,(p,n)) for some

n, and $|\lambda| < |\Delta\varphi_C|$, p$\not\epsilon$dom Π in S (4)+(27)+Lemma 5.2-1+Def. 2.3-1

(40) $\not\exists i$: $\varphi_1 \epsilon SB_\Omega(ID,i)$ (37)+(38)+(39)

(41) There is a Select operator S and integers j and i such that

$\varphi_1 \epsilon SB_\Omega(S,j,i)$ (36)+(40)+(20)+Def. 3.3-13

(42) The jth tokens to appear on S's number-i output arcs in Ω have value

(p,R) or (p,W) and that appearance does not follow the appearance

of the token removed by φ_1 from d_1's primary input arc

         (41)+(28)+(27)+(20)+Def. 3.3-13

(43) d_1 is enabled in $S' \cdot \theta'$ and d_2 is enabled in $S \cdot \theta$

          (4)+(5)+(10)+Def. 2.3-1

(44) There is a token on d_1's primary input arc in $S' \cdot \theta'$

         (43)+Defs. 3.3-6+2.1-4

(45) $S' \cdot \theta'$ and $S \cdot \theta$ are equal states   (4)+(5)+(10)+Thm. 7.1-2

(46) There is a token on d_1's primary input arc b in $S \cdot \theta$

         (44)+(45)+Defs. 7.1-2+3.4-1

(47) b is not an output arc of d_2 (43)+(46)+Defs. 3.3-6+2.1-4

(48) b is not a data-output arc of a Select operator which is in a pool

in $S \cdot \theta$ (46)+Cor. 7.1-1

(49) No token can appear on b in the transition from $S \cdot \theta$ to $S \cdot \theta\varphi_2$; i.e.,

the token removed by φ_1 is on d_1's primary input arc in $S \cdot \theta$

         (46)+(47)+(48)+Defs. 3.3-9+2.1-5

(50) There are prefixes of θ such that there are tokens of value (p,R) on S's output arcs after those prefixes. Let $\Delta\varphi_A$ be the shortest of these; i.e., there are tokens of value (p,R) on S's output arcs in $S \cdot \Delta\varphi_A$ but not in $S \cdot \Delta$ (49)+(42)+Def. 3.3-9

(51) Let A be the actor of which φ_A is a firing. Then S is in Q(p) in Fire($S \cdot \Delta$,A) and there is no token with value (p,W) on an arc in $S \cdot \Delta\varphi_A$ (50)+Def. 3.3-9

(52) There is a prefix $\Xi\varphi_S$ of $\Delta\varphi_A$ in which φ_S is a firing of S such that, for $S \cdot \Xi = (\Gamma,U)$, there are tokens of value p on S's output arcs in $\text{Standard}_\Gamma((\text{Strip}(\Gamma,S),U),S)$ (51)+Defs. 3.3-5+3.3-9

(53) There is some node n such that $\Pi(p) \in SM(n)$ in $S \cdot \Xi$ (51)+Def. 2.2-5

(54) Let S_S be the standard state corresponding to S. Then Ξ is a firing sequence starting in S_S and $S_S \cdot \Xi \mu S \cdot \Xi$ Thm. 7.1-1

(55) The heap in $S \cdot \Xi$ is identical to that in $S_S \cdot \Xi$ (54)+Def. 7.1-1

(56) There is some node n such that $\Pi(p)$ is in SM(n) in $S_S \cdot \Xi$ (55)+(53)

(57) $p \in \text{dom } \Pi$ in $S_S \cdot \Xi$, hence $p \in \text{dom } \Pi$ in $S \cdot \Xi$ (55)+(56)+Thm. 2.2-1

(58) Δ does not contain the nth firing of C $\Rightarrow$ Ξ does not contain the nth firing of C $\Rightarrow$ $p \notin \text{dom } \Pi$ in $S \cdot \Xi$ (52)+(27)+(4)+Lemma 5.2-1+Def. 2.3-1

(59) Δ does contain the nth firing of C   (58)+(57)

(60) There is a prefix $X\varphi$ of Ω with $|\Delta| < |X| < |\theta|$ such that there is no token with value (p,W) in $S \cdot X$ but there is one in $S \cdot X\varphi$ (51)+(27)+(10)

(61) That token can appear only on an output arc of a Copy or pI operator (not a Select), and then only if φ is a firing of that operator   Defs. 3.3-9+2.2-5

(62) φ is a pI firing $\Rightarrow$ there is a token of value (p,W) on an arc in $S \cdot \chi$

Defs. 3.3-9+2.2-4+3.3-8

(63) φ is a Copy firing which outputs (p,W) but is not the nth firing of

C   (61)+(62)+(60)+(59)

Since (1) leads to a contradiction between (63) and (36), (1) is false

(64) (Int,J) satisfies the Freedom-from-conflict Axiom

$\triangle$

7.2.3 Commutativity

This axiom asserts that for any computation $\alpha gf\delta$ in a job $J \in J$ such that $\alpha fg\delta$ is also in J, $ET_J(\alpha fg\delta) = ET_J(\alpha gf\delta)$. For any transfer t in $ET_J(\alpha gf\delta)$, there is an entry h with T(h) = t such that $\alpha gf\delta h$ is in J. By construction of J, there is a γ such that $\beta = \alpha gf\delta h\gamma$ is in J_{S_1,Ω_1} for some initial modified state S_1 and halted firing sequence Ω_1 starting in S_1. There is also an initial modified state S_2 equal to S_1 and halted firing sequence Ω_2 starting in S_2 such that $\alpha fg\delta$ is a prefix of some β' in J_{S_2,Ω_2}. The thrust of the proof is to show that there is a halted firing sequence Ω' starting in S_1 such that $\alpha fg\delta h\gamma$ is in $J_{S_1,\Omega'}$; it then follows that $\alpha fg\delta h$ is in J, so T(h) = t is in $ET_J(\alpha fg\delta)$. By symmetry, every transfer in $ET_J(\alpha gf\delta)$ is in $ET_J(\alpha fg\delta)$, so $ET_J(\alpha gf\delta) = ET_J(\alpha fg\delta)$.

For $\alpha fg\delta h\gamma$ to be in $J_{S_1,\Omega'}$, it must be causal (Definition 4.3-5). Both β' and β are known to be causal. For any prefix ϵk of $\alpha fg\delta h\gamma$ in which k is an output entry of execution e, if ϵk is a prefix of $\alpha fg\delta$, then e is initiated in ϵ by the causality of β'. Otherwise, there is an ι such that $\epsilon k = \alpha fg\delta\iota k$. This implies that $\alpha gf\delta\iota k$ is a prefix of β, so the initiating entry of e is in $\alpha gf\delta\iota$. Therefore, e is also initiated in $\alpha fg\delta\iota = \epsilon$, so $\alpha fg\delta h\gamma$ is causal.

The remainder of the proof is divided into two cases.

Case I: f and g do not initiate distinct executions of actors in P.

In this case, $\Omega' = \Omega_1$. Since β is a permutation of $\eta(S_1,\Omega_1)$, $\alpha fg\delta h\gamma$ is a (causal) permutation of $\eta(S_1,\Omega')$. If neither of f and g initiates an execution in αgf, then neither initiates an execution in αfg, so $\Phi(\alpha fg) = \Phi(\alpha) = \Phi(\alpha gf)$. If one of f and g initiates an execution $e = Ex(d,k)$ in αgf, then one of f and g initiates e in αfg (if they are both input entries to e, then the initiating entry in either computation is the later one). Then $\Phi(\alpha fg) = \Phi(\alpha gf) = \Phi(\alpha)\varphi$, where φ is a firing of d. It is known that $\Phi(\beta) = \Phi(\alpha gf\delta h\gamma)$ is the reduction of Ω_1. The same firings which follow $\Phi(\alpha gf)$ in $\Phi(\alpha gf\delta h\gamma)$ also follow $\Phi(\alpha fg)$ in $\Phi(\alpha fg\delta h\gamma)$, and do so in the same order. Since $\Phi(\alpha gf) = \Phi(\alpha fg)$, $\Phi(\alpha fg\delta h\gamma) = \Phi(\alpha gf\delta h\gamma)$, which is $\Phi(\beta)$, the reduction of $\Omega_1 = \Omega'$ (Lemma 7.2-4 below).

The final condition which must be met for $\alpha fg\delta h\gamma$ to be in $J_{S_1,\Omega'}$ concerns each of its prefixes εk. First the following observation is made about εk and the prefix Δ of Ω' whose reduction is $\Phi(\varepsilon)$: If εk is a prefix of $\alpha fg\delta$, then ε is a prefix of β', so letting Δ_2 be the prefix of Ω_2 whose reduction is $\Phi(\varepsilon)$, Δ_2 equals Δ; hence $S_2 \cdot \Delta_2$ equals $S_1 \cdot \Delta$. Otherwise, as noted above, there is a prefix $\varepsilon'k$ of $\beta = \alpha gf\delta h\gamma$ such that $\Phi(\varepsilon') = \Phi(\varepsilon)$; therefore, letting Δ_1 be the prefix of Ω_1 whose reduction is $\Phi(\varepsilon')$, Δ_1 equals Δ, so $S_1 \cdot \Delta_1$ equals $S_1 \cdot \Delta$. I.e., $\alpha fg\delta h\gamma$ is a causal permutation of $\eta(S_1,\Omega')$, $\Phi(\alpha fg\delta h\gamma)$ is the reduction of Ω', and for each prefix εk of $\alpha fg\delta h\gamma$, letting Δ be the prefix of Ω' whose reduction is $\Phi(\varepsilon)$, there is an initial state S' equal to S and a halted firing sequence Ω starting in S', and there is a prefix $\varepsilon'k$ of some $\iota \in J_{S',\Omega}$ such that, letting Δ' be the prefix of Ω whose reduction is $\Phi(\varepsilon')$, $S' \cdot \Delta'$ equals $S_1 \cdot \Delta$.

Completing the proof in this case requires considering the destination $\text{Dst}(\text{Ex}(d,m),j)$ in $T(k)$. If $d \notin \text{DL}$, then because $\iota \in J_{S',\Omega'}$, d is enabled in $S' \cdot \Delta'$, and if it is a merge gate, and its number-j input arc is its T (F) input arc, then its control input arc holds a <u>true</u> (<u>false</u>) token in $S' \cdot \Delta'$. Because $S_1 \cdot \Delta$ equals $S' \cdot \Delta'$, d is enabled in $S_1 \cdot \Delta$ with the same input tokens (Corollary 7.1-2). If $d \in \text{DL}$ and $d = (c,n)$, then there is a token in $S' \cdot \Delta'$ on the arc b which is either the number-n program output arc of P if $c = $ "OD", or the number-n input arc of the actor labelled c; furthermore, if $c \neq$ "OD", there is no firing sequence starting in $S' \cdot \Delta'$ which contains a firing of c. Since $S_1 \cdot \Delta$ equals $S' \cdot \Delta'$, there is a token on b in $S_1 \cdot \Delta$, and any firing sequence starting in $S_1 \cdot \Delta$ is a firing sequence starting in $S' \cdot \Delta'$ (Corollary 7.1-2), and so does not contain a firing of c. These conclusions, together with the fact that $\alpha\text{fg}\delta\text{h}\gamma$ is a causal permutation of $\eta(S_1,\Omega')$ and $\Phi(\alpha\text{fg}\delta\text{h}\gamma)$ is the reduction of Ω', mean that $\alpha\text{fg}\delta\text{h}\gamma$ is in $J_{S_1,\Omega'}$.

(Reasoning similar to that in the final paragraph above is used in the proof that every expansion satisfies the Persistence Axiom. For efficiency, the results needed in both proofs are combined into one lemma. Unfortunately, a complete understanding of that lemma requires considerations unique to persistence. To avoid a disruptive digression here, the presentation of the lemma [Lemma 7.2-8] is postponed until after the proof of Commutativity, in which it is used; this does not, however, introduce any circularity.)

Case II: f and g initiate executions $e_1 = \text{Ex}(d_1,k_1)$ and $e_2 = \text{Ex}(d_2,k_2)$.

There are prefixes $\theta\varphi_2\varphi_1$ of Ω_1 and $\theta'\varphi_1'\varphi_2'$ of Ω_2 whose reductions are $\Phi(\alpha\text{gf})$ and $\Phi(\alpha\text{fg})$ and θ' equals θ. Furthermore, φ_1 and φ_2 potentially

interfere iff, in αgf, $Ent(e_1,1)$ and $Ent(e_2,1)$ are in the same access history and $e_1 \in R(e_2)$ (Lemma 7.2-3). By freedom-from-conflict, it is not true that $Ent(e_1,1)$ and $Ent(e_2,1)$ are in the same access history and $e_1 \in R(e_2)$. Therefore, φ_1 and φ_2 do not potentially interfere in $\theta\varphi_2\varphi_1$. Since φ_1 and φ_1' (φ_2 and φ_2') are firings of the same actor, $\theta\varphi_1\varphi_2$ equals $\theta'\varphi_1'\varphi_2'$, which is a firing sequence starting in S_2. S_1 equals S_2, so $\theta\varphi_1\varphi_2$ is a firing sequence starting in S_1 (Corollary 7.1-2). I.e., there are two firing sequences $\theta\varphi_1\varphi_2$ and $\theta\varphi_2\varphi_1$ starting in S_1 such that φ_1 and φ_2 do not potentially interfere in $\theta\varphi_2\varphi_1$.

For S_1' the initial standard state corresponding to S_1, $\theta\varphi_2\varphi_1$ and $\theta\varphi_1\varphi_2$ are firing sequences starting in S_1', $S_1' \cdot \theta\varphi_2\varphi_1 \mu S_1 \cdot \theta\varphi_2\varphi_1$ and $S_1' \cdot \theta\varphi_1\varphi_2 \mu S_1 \cdot \theta\varphi_1\varphi_2$. Since φ_1 and φ_2 do not potentially interfere, an earlier argument for standard states (Theorem 3.1-1) applies, holding that φ_1 and φ_2 do not interfere; i.e., $S_1' \cdot \theta\varphi_2\varphi_1$ and $S_1' \cdot \theta\varphi_1\varphi_2$ are identical standard states. Modified states differ from standard states in two regards: tagged pointers (p,R) and (p,W) take the places of simple pointers as the values of tokens, and there is a third, pool component in a modified state. By the congruency relation μ, the heap in $S_1 \cdot \theta\varphi_2\varphi_1$ is identical to that in $S_1' \cdot \theta\varphi_2\varphi_1$, which is identical to that in $S_1' \cdot \theta\varphi_1\varphi_2$, which is identical to that in $S_1 \cdot \theta\varphi_1\varphi_2$. Similarly, each arc holds a non-pointer-valued token in $S_1 \cdot \theta\varphi_2\varphi_1$ iff it holds a token of the same value in $S_1 \cdot \theta\varphi_1\varphi_2$. Finally, if the pool components are identical, each arc holds a token of value (p,R) or (p,W) in $S_1 \cdot \theta\varphi_2\varphi_1$ iff it holds a token of value (p,R) or (p,W) in $S_1 \cdot \theta\varphi_1\varphi_2$.

Lemma 7.2-5 below proves sufficient conditions under which, for any arc holding tokens in two different states $S \cdot \Delta_1$ and $S \cdot \Delta_2$ whose values are

tagged pointers, either both are read pointers or both are write pointers. The only arcs which can hold pointer-valued tokens are program input arcs and output arcs of Copy, Select, or pI operators (including gates). In all but the last case, the arc either always holds a read pointer or always holds a write pointer. For a pI actor d, the kth firings of d in Δ_1 and Δ_2 output tokens removed from the same arc b if [d is a gate $\Rightarrow$ those firings had identical control inputs]. If there is a k' such that the last firing of the actor of which b is an output arc is the k'^{th} in both Δ_1 and Δ_2, then a simple inductive argument shows that the kth firings in Δ_1 and Δ_2 of any pI actor either both output read pointers or both output write pointers. Then if every actor fires the same number of times in Δ_1 and Δ_2, the values of the tokens on any arc in $S \cdot \Delta_1$ and $S \cdot \Delta_2$ are either both read pointers or both write pointers.

From the preceding two paragraphs, the configuration and heap components in $S_1 \cdot \theta \varphi_2 \varphi_1$ and $S_1 \cdot \theta \varphi_1 \varphi_2$ are identical if the pool components are. A careful accounting shows that for each pointer p, the number of tokens with value (p,W) is the same in both states. For each label S which is in Q(p) in $S_1 \cdot \theta$, then, S will have been removed in $S_1 \cdot \theta \varphi_2 \varphi_1$ iff there are zero tokens of value (p,W) in $S_1 \cdot \theta \varphi_2 \varphi_1$ and $S_1 \cdot \theta \varphi_1 \varphi_2$ iff S has been removed from Q(p) in $S_1 \cdot \theta \varphi_1 \varphi_2$. If either φ_1 or φ_2 is a firing of S, it has the same inputs in both $\theta \varphi_2 \varphi_1$ and $\theta \varphi_1 \varphi_2$ and fires in the same heap, and so will try to output the same pointer. Thus, S will get added to the same pool in either firing sequence, and will have been removed from that pool in $S_1 \cdot \theta \varphi_2 \varphi_1$ iff it has been removed in $S_1 \cdot \theta \varphi_1 \varphi_2$. Therefore, the pool components of $S_1 \cdot \theta \varphi_2 \varphi_1$ and $S_1 \cdot \theta \varphi_1 \varphi_2$ are identical, so the states themselves are identical (Theorem 7.2-2 below).

The halted firing sequence Ω_1 has $\theta\varphi_2\varphi_1$ as a prefix, and so may be written as $\theta\varphi_2\varphi_1\Xi$. Since $S_1\cdot\theta\varphi_2\varphi_1$ and $S_1\cdot\theta\varphi_1\varphi_2$ are identical, it is evident that for any prefixes Δ_1 of Ω_1 and Δ' of $\theta\varphi_1\varphi_2\Xi$ such that $|\Delta_1| = |\Delta'| \geq |\theta\varphi_2\varphi_1|$, $S_1\cdot\Delta_1$ and $S_1\cdot\Delta'$ are identical; furthermore, $\Omega' = \theta\varphi_1\varphi_2\Xi$ is also a halted firing sequence starting in S_1. $\Phi(\alpha gf)$ is the reduction of $\theta\varphi_2\varphi_1$, $\Phi(\alpha fg)$ is the reduction of $\theta\varphi_1\varphi_2$, and $\Phi(\alpha gf\delta h\gamma)$ is $\Phi(\beta)$, the reduction of $\Omega_1 = \theta\varphi_2\varphi_1\Xi$; hence, $\Phi(\alpha fg\delta h\gamma)$ is the reduction of $\theta\varphi_1\varphi_2\Xi = \Omega'$ (Lemma 7.2-4). Because both d_1 and d_2 are enabled in $S_1\cdot\theta$, no output arc of one is an input arc of the other, so any token removed from an arc b by φ_1 or φ_2 in either $\theta\varphi_2\varphi_1$ or $\theta\varphi_1\varphi_2$ is on b in $S_1\cdot\theta$. Thus φ_1 removes the same tokens from the same arcs in $\theta\varphi_2\varphi_1$ and $\theta\varphi_1\varphi_2$; φ_2 does also. Since every other firing fires in identical states, for any d and k, the kth firings of d in Ω_1 and Ω' both remove the same tokens from the same arcs. Furthermore, for any such arc which is an output arc of an actor d', the kth firings of d in Ω_1 and Ω' are preceded by the same number of firings of d'. Finally, each arc has a token left in the final state $S_1\cdot\Omega_1$ iff it has an identical token in $S_1\cdot\Omega'$. By Algorithm 3.4-1, then, $\eta(S_1,\Omega')$ has the same set of entries as $\eta(S_1,\Omega_1)$ (Lemma 7.2-6 below).

Thus far, it has been seen that $\alpha fg\delta h\gamma$ is a permutation of $\alpha gf\delta h\gamma$, which is a permutation of $\eta(S_1,\Omega_1)$, which is a permutation of $\eta(S_1,\Omega')$, and that $\Phi(\alpha fg\delta h\gamma)$ is the reduction of Ω'. The following is sufficient to prove that $\alpha fg\delta h\gamma$ is in $\eta(S_1,\Omega')$: For each prefix ϵk of $\alpha fg\delta h\gamma$, letting κ be the prefix of κ' whose reduction is $\Phi(\epsilon)$, there is an initial state S', and a halted firing sequence Ω starting in S', and a computation in J, such that, letting Λ'

be the prefix of Ω whose reduction is $\Phi(\epsilon')$, $S' \cdot \Delta'$ equals $S_1 \cdot \Delta$

(Lemma 7.2-8). If ϵk is a prefix of $\alpha f g \delta$, then, as in Case I, ϵ is a

prefix of β', so $S' = S_2$ and $\Omega = \Omega_2$. Otherwise, there is an ι such that

$\epsilon = \alpha f g \iota$. Since $\Phi(\alpha f g)$ is the reduction of $\theta \varphi_1 \varphi_2$, there is a X such that

$\Delta = \theta \varphi_1 \varphi_2 X$. Since $\Phi(\alpha g f)$ is the reduction of $\theta \varphi_2 \varphi_1$, there is a prefix

$\epsilon' k = \alpha g f \iota k$ of $\alpha g f \delta h \gamma \in J_{S_1, \Omega_1}$ such that, for Δ' the prefix of Ω_1 whose

reduction is $\Phi(\epsilon')$, $\Delta' = \theta \varphi_2 \varphi_1 X$ (Lemma 7.2-4). $|\Delta'| = |\Delta| \geq |\theta \varphi_2 \varphi_1|$, so

$S_1 \cdot \Delta'$ and $S_1 \cdot \Delta$ are identical, hence equal. Therefore, $\alpha f g \delta h \gamma$ is in $J_{S_1, \Omega'}$.

The various lemmas and theorems for which informal proofs have just

been given are now presented in their precise forms.

Lemma 7.2-4 Let α_1, α_2, and β be any three sequences of entries such that

α_2 is a permutation of α_1. Let Int = (St, I, IE) be an interpretation. Let

θ_1 and θ_2 be such that, for i=1,2, the firing sequence $\Phi(\alpha_i)$ reconstructed

from α_i with respect to Int is the reduction of θ_i. Then for any Δ,

$\Phi(\alpha_1 \beta)$ is the reduction of $\theta_1 \Delta \Rightarrow \Phi(\alpha_2 \beta)$ is the reduction of $\theta_2 \Delta$.

Proof: By induction on the length of β. All initiations and reconstruc-

tions are with respect to Int.

Basis: $|\beta| = 0$.

  (1) Since $\alpha_1 \beta = \alpha_1$, for any Δ, $\Phi(\alpha_1 \beta)$ is the reduction of $\theta_1 \Delta \Rightarrow \Phi(\alpha_1)$ is

     the reduction of $\theta_1 \Delta \Rightarrow$ the reduction of θ_1 equals the reduction of

     $\theta_1 \Delta \Rightarrow |\Delta| = 0$ Def. 2.4-5

  (2) $\Rightarrow \Phi(\alpha_2 \beta) = \Phi(\alpha_2)$ is the reduction of $\theta_2 \Delta = \theta_2$

Induction step: Assume the Lemma is true for any β of length $n \geq 0$, and

consider $\beta = \gamma f$ of length n+1.

(3) Let $e = \text{Ex}(d,k)$ be the target of f. f is not the initiating entry

of e in $\alpha_1\beta$ iff there are fewer than $\text{In}(l(d))$ input entries to e

in $\alpha_1\beta$ iff there are fewer than $\text{In}(l(d))$ input entries to e in

$\alpha_2\beta$ iff f is not the initiating entry of e in $\alpha_2\beta$ Def. 4.2-6

(4) f is not the initiating entry of e in $\alpha_1\beta \Rightarrow \Phi(\alpha_1\beta) = \Phi(\alpha_1\gamma) \wedge$

     $\Phi(\alpha_2\beta) = \Phi(\alpha_2\gamma)$ (3)+Def. 4.3-4

(5) $\Rightarrow$ [for any Δ, $\Phi(\alpha_1\beta)$ is the reduction of $\theta_1\Delta \Rightarrow \Phi(\alpha_1\gamma)$ is the

     reduction of $\theta_1\Delta \Rightarrow \Phi(\alpha_2\gamma)$ is the reduction of $\theta_2\Delta$ ind. hyp.

(6) $\Rightarrow \Phi(\alpha_2\beta)$ is the reduction of $\theta_2\Delta$]   (4)

(7) f is the initiating entry of e in $\alpha_1\beta \Rightarrow \Phi(\alpha_1\beta) = \Phi(\alpha_1\gamma)\varphi'$ and

     $\Phi(\alpha_2\beta) = \Phi(\alpha_2\gamma)\varphi'$, where φ' is a firing of d (3)+Def. 4.3-4

(8) $\Rightarrow$ [for any Δ, $\Phi(\alpha_1\beta)$ is the reduction of $\theta_1\Delta \Rightarrow \theta_1\Delta = \theta_1\Xi\varphi$, where φ

     is a firing of d Def. 2.4-5

(9) $\Rightarrow$ the reduction of $\theta_1\Xi$ is that prefix of the reduction of $\theta_1\Delta$ which

     is one firing shorter than the reduction of $\theta_1\Delta$ Def. 2.4-5

(10) $\Rightarrow$ the reduction of $\theta_1\Xi$ is $\Phi(\alpha_1\gamma)$ (8)+(7)

(11) $\Rightarrow \Phi(\alpha_2\gamma)$ is the reduction of $\theta_2\Xi$ ind. hyp.

(12) $\Rightarrow$ the reduction of $\theta_2\Delta$ is $\Phi(\alpha_2\gamma)\varphi'$, where φ' is a firing of d

     Def. 2.4-5

(13) $\Rightarrow$ the reduction of $\theta_2\Delta$ is $\Phi(\alpha_2\beta)$]   (8)+(7)

(14) For any Δ, $\varphi(\alpha_1\beta)$ is the reduction of $\theta_1\Delta \Rightarrow \Phi(\alpha_2\beta)$ is the reduction

     of $\theta_2\Delta$ (4)+(5)+(6)+(7)+(8)+(13)

$\triangle$

Lemma 7.2-5 Let S_1 and S_2 be any two equal initial modified states for

the same program P. Let Ω_1 and Ω_2 be two firing sequences starting in S_1

and S_2 respectively such that

(1) for each actor d in P, there are the same number of firings of d

   in both Ω_1 and Ω_2,

(2) for each gate d in P and each k, the kth firings of d in Ω_1 and Ω_2

   remove control tokens of the same value, and

(3) for any two actors d and d', and for any k, there is a k' such that

   if the kth firings of d in Ω_1 and Ω_2 remove tokens from output arcs

   of d', then those firings both are preceded by k' firings of d'.

Then for any arc in P which holds tokens of pointer value in $S_1 \cdot \Omega_1$ and

$S_2 \cdot \Omega_2$, either both are read pointers or both are write pointers.

Proof: (The straight-forward inductive proof has already been outlined

above; the rigorous treatment has been removed to Appendix E.)

$\triangle$

Theorem 7.2-2 Given any L_D program P, let $\theta\varphi_2\varphi_1$ be any firing sequence

starting in any initial modified state S of P, such that $\theta\varphi_1\varphi_2$ is also a

firing sequence starting in S. If φ_1 and φ_2 do not potentially interfere

in $\theta\varphi_2\varphi_1$, then $S \cdot \theta\varphi_2\varphi_1$ and $S \cdot \theta\varphi_1\varphi_2$ are identical states.

Proof:

   (1) θ is a firing sequence starting in S Def. 2.3-1

   (2) Let S' be the initial standard state corresponding to S. Then θ,

      $\theta\varphi_2\varphi_1$, and $\theta\varphi_1\varphi_2$ are all firing sequences starting in S',

      $S' \cdot \theta \mu S \cdot \theta$, $S' \cdot \theta\varphi_2\varphi_1 \mu S \cdot \theta\varphi_2\varphi_1$, and $S' \cdot \theta\varphi_1\varphi_2 \mu S \cdot \theta\varphi_1\varphi_2$ (1)+Thm. 7.1-1

   (3) φ_1 and φ_2 do not interfere in $\theta\varphi_2\varphi_1$ Thm. 3.1-1

   (4) $S' \cdot \theta\varphi_2\varphi_1$ and $S' \cdot \theta\varphi_1\varphi_2$ are identical states   (3)+Def. 3.1-1

   (5) Let d_1 and d_2 be the actors of which φ_1 and φ_2 are firings. Then

      both actors are enabled in $S \cdot \theta$ Def. 2.3-1

(6) If either is a gate, its control input arc has a token in $S \cdot \theta$ and

so is not an output arc of the other   (5)+Defs. 3.3-6+2.1-4

(7) If d_1 (d_2) is a gate, the control token input by φ_1 (φ_2) in either

$\theta \varphi_2 \varphi_1$ or $\theta \varphi_1 \varphi_2$ is the token on the control input arc in $S \cdot \theta$

          (6)+Defs. 3.3-9+3.3-7+2.1-5

(8) The sets of input arcs from which φ_1 (φ_2) removes tokens is the

same in both $\theta \varphi_2 \varphi_1$ and $\theta \varphi_1 \varphi_2$ (7)+Defs. 3.3-9+2.1-5

(9) All of those arcs have tokens in $S \cdot \theta$ (5)+(7)+Defs. 3.3-6+2.1-4

(10) None of those arcs is an output arc of either d_1 or d_2

          (5)+(9)+Defs. 3.3-6+2.1-4

(11) All of the tokens removed by φ_1 (φ_2) in either $\theta \varphi_2 \varphi_1$ or $\theta \varphi_1 \varphi_2$ are

on the arcs from which they are removed in $S \cdot \theta$ (10)+Def. 2.1-5

(12) For any pointer p and any arc b, there is a token with value (p,W)

on b in $S \cdot \theta$ but no such token in $S \cdot \theta \varphi_2 \varphi_1$ iff there is a token with

value (p,W) on b in $S \cdot \theta$ but no such token in $S \cdot \theta \varphi_1 \varphi_2$ (8)+(11)

(13) For each arc b, b holds a token of value (p,W) in $S \cdot \theta \Rightarrow$ b is not

an output arc of either d_1 or d_2 (5)+Defs. 3.3-6+2.1-4

(14) $\Rightarrow$ if b holds a token in either $S \cdot \theta \varphi_2 \varphi_1$ or $S \cdot \theta \varphi_1 \varphi_2$, its value is

(p,W)   Defs. 3.3-9+2.1-5

(15) Tokens with write pointers as values can be placed on the output

arcs only of Copy or pI operators   Defs. 3.3-9+3.3-7+2.2-5

(16) Neither φ_1 nor φ_2 is a Copy firing which outputs p $\Rightarrow$ for each arc

b, there is a token with value (p,W) on b in $S \cdot \theta \varphi_1 \varphi_2$ but no such

token in $S \cdot \theta$ iff b is an output arc of d_1 or d_2, that actor is a

pI operator, and if it is a gate, the control input to φ_1 or φ_2

in $\theta \varphi_1 \varphi_2$ is such that tokens are placed on all output arcs and

their values equal the value of the token removed from an input

arc a by φ_1 or φ_2 in $\theta\varphi_1\varphi_2$, which token has value (p,W)

(15)+Defs. 3.3-9+2.1-5+2.2-4

(17) iff b is an output arc of d_1 or d_2, that actor is a pI operator,

and if it is a gate, the control input to φ_1 or φ_2 in $\theta\varphi_2\varphi_1$ is

such that tokens are placed on all output arcs and their values

equal the value of the token removed from input arc a by φ_1 or

φ_2 in $\theta\varphi_2\varphi_1$, which token has value (p,W)   (7)+(11)

(18) iff there is a token with value (p,W) on b in $S \cdot \theta\varphi_2\varphi_1$ but no such

token on b in $S \cdot \theta$ (15)+Defs. 3.3-9+2.1-5+2.2-4

(19) The number of tokens with value (p,W) in $S \cdot \theta\varphi_2\varphi_1$ (or $S \cdot \theta\varphi_1\varphi_2$) is

the number of tokens with value (p,W) in $S \cdot \theta$, minus the number of

arcs which hold tokens of value (p,W) in $S \cdot \theta$ but not in $S \cdot \theta\varphi_2\varphi_1$

(or $S \cdot \theta\varphi_1\varphi_2$), plus the number of arcs which hold no tokens in $S \cdot \theta$

but a token of value (p,W) in $S \cdot \theta\varphi_2\varphi_1$ (or $S \cdot \theta\varphi_1\varphi_2$)   (13)+(14)

(20) If neither φ_1 or φ_2 is a Copy firing which outputs p, then the

number of tokens with value (p,W) in $S \cdot \theta\varphi_2\varphi_1$ equals the number of

tokens with value (p,W) in $S \cdot \theta\varphi_1\varphi_2$ (19)+(12)+(16)+(18)

(21) For any arc b, b holds a token of value (p,W) in $S \cdot \theta\varphi_2\varphi_1$ ($S \cdot \theta\varphi_1\varphi_2$)

but not in $S \cdot \theta\varphi_2$ ($S \cdot \theta\varphi_1$) $\Rightarrow \varphi_1$ (φ_2) is a Copy firing which outputs

p, or φ_1 (φ_2) is a pI firing and some input arc of d_1 (d_2) holds

a token of value (p,W) in $S \cdot \theta\varphi_2$ ($S \cdot \theta\varphi_1$)   (15)+Def. 2.2-4

(22) For any Select operator S not equal to d_1 or d_2, and any pointer

p, $S \in Q(p)$ in $S \cdot \theta \Rightarrow$ there are tokens of value p on S's output

arcs in $S' \cdot \theta$ (2)+Def. 7.1-1

(23) $\Rightarrow$ p$\in$dom Π in $S' \cdot \theta$ Thm. 2.2-1

(24) $\Rightarrow$ p$\in$dom Π in $S\cdot\theta$ (2)+Def. 7.1-1

(25) $\Rightarrow$ neither φ_1 nor φ_2 is a Copy firing which outputs p   Lemma 5.2-1

(26) $\Rightarrow$ if there are zero tokens with value (p,W) in $S\cdot\theta\varphi_2$ $(S\cdot\theta\varphi_1)$,

then there are zero tokens with value (p,W) in $S\cdot\theta\varphi_2\varphi_1$ $(S\cdot\theta\varphi_1\varphi_2)$

                                                (21)

(27) S$\in$Q(p) in $S\cdot\theta$ $\Rightarrow$ S$\notin$Q(p) in $S\cdot\theta\varphi_2\varphi_1$ $(S\cdot\theta\varphi_1\varphi_2)$ iff there are zero

tokens with value (p,W) in either $S\cdot\theta\varphi_2$ or $S\cdot\theta\varphi_2\varphi_1$ $(S\cdot\theta\varphi_1$ or

$S\cdot\theta\varphi_1\varphi_2)$ Def. 3.3-9

(28) iff there are zero tokens with value (p,W) in $S\cdot\theta\varphi_2\varphi_1$ $(S\cdot\theta\varphi_1\varphi_2)$

                                             (22)+(26)

(29) For any Select operator S not equal to d_1 or d_2, and any pointer p,

S$\in$Q(p) in either $S\cdot\theta\varphi_2\varphi_1$ or $S\cdot\theta\varphi_1\varphi_2$ $\Rightarrow$ S$\in$Q(p) in $S\cdot\theta$ Def. 3.3-9

(30) $\Rightarrow$ S$\in$Q(p) in $S\cdot\theta\varphi_2\varphi_1$ iff S$\in$Q(p) in $S\cdot\theta\varphi_1\varphi_2$ (27)+(28)+(22)+(25)+(20)

(31) Assume, say, d_1 is a Select operator. Let Δ be any firing sequence

starting in S such that d_1 is enabled in $S\cdot\Delta$; i.e., $\Delta\varphi$, where φ

is a firing of d_1, is a firing sequence starting in S Def. 2.3-1

(32) Δ and $\Delta\varphi$ are firing sequences starting in S', $S'\cdot\Delta\mu S\cdot\Delta$ and

$S'\cdot\Delta\varphi\mu S\cdot\Delta\varphi$ (31)+Thm. 7.1-1

(33) Let $S\cdot\Delta$ be (Γ,U). Then U is the heap in $S'\cdot\Delta$ (32)+Def. 7.1-1

(34) Let q and s be the values of the tokens on d_1's number-1 and number-

2 input arcs in Strip(Γ,d_1). Then those arcs have tokens of value

(q,R) or (q,W) and s in $S\cdot\Delta$ (33)+Def. 3.3-8

(35) Those arcs have tokens of value q and s in $S'\cdot\Delta$ (34)+(32)+Def. 7.1-1

(36) There are tokens of value p on d_1's output arcs in

Standard$_\Gamma$((Strip(Γ,d_1),U),d_1) iff there is a pair (s,Π(p)) in

SM(Π(q)) in U   (34)+Defs. 3.3-7+2.2-5

(37) iff there are tokens of value p on d_1's output arcs in $S' \cdot \Delta\varphi$

(35)+(33)+Def. 2.2-5

(38) Letting $S \cdot \theta$ be (Γ, U, Q), for any p, $d_1 \in Q(p)$ in $\text{Fire}(S \cdot \theta, d_1)$ iff

there are tokens of value p on d_1's output arcs in

$\text{Standard}_\Gamma((\text{Strip}(\Gamma, d_1), U), d_1)$ (31)+Def. 3.3-9

(39) iff there are tokens of value p on d_1's output arcs in $S' \cdot \theta\varphi_1$

(31)+(33)+(36)+(37)

(40) iff there are tokens of value p on those arcs in $S' \cdot \theta\varphi_1\varphi_2$ (8)+(10)

(41) iff there are tokens of value p on d_1's output arcs in $S' \cdot \theta\varphi_2\varphi_1$ (4)

(42) iff, letting $S \cdot \theta\varphi_2$ be (Γ', U', Q'), there are tokens of value p on

d_1's output arcs in $\text{Standard}_\Gamma((\text{Strip}(\Gamma', d_1), U'), d_1)$

(31)+(33)+(36)+(37)

(43) iff $d_1 \in Q(p)$ in $\text{Fire}(S \cdot \theta\varphi_2, d_1)$ (31)+Def. 3.3-9

(44) There are tokens on d_1's output arcs of value p in $S' \cdot \theta\varphi_1 \Rightarrow$

$p \in \text{dom } \Pi$ in that state   Thm. 2.2-1

(45) $\Rightarrow \varphi_2$ is not a Copy firing which outputs p, nor is φ_1(31)+Lemma 5.2-1

(46) For any p, $d_1 \in Q(p)$ in $S \cdot \theta\varphi_1\varphi_2$ iff $d_1 \in Q(p)$ in $\text{Fire}(S \cdot \theta, d_1)$ and

there are not zero tokens of value (p,W) in $S \cdot \theta\varphi_1$ and there are

not zero tokens of value (p,W) in $S \cdot \theta\varphi_1\varphi_2$ Def. 3.3-9

(47) iff $d_1 \in Q(p)$ in $\text{Fire}(S \cdot \theta, d_1)$ and there are not zero tokens of value

(p,W) in $S \cdot \theta\varphi_1\varphi_2$ (38)+(39)+(44)+(45)+(25)+(26)

(48) iff $d_1 \in Q(p)$ in $\text{Fire}(S \cdot \theta\varphi_2, d_1)$ and there are not zero tokens of value

(p,W) in $S \cdot \theta\varphi_2\varphi_1$ (38)+(43)+(39)+(44)+(45)+(20)+(25)+(26)

(49) iff $d_1 \in Q(p)$ in $S \cdot \theta\varphi_2\varphi_1$ Def. 3.3-9

By symmetry,

(50) d_2 is a Select $\Rightarrow \forall p$, $d_2 \in Q(p)$ in $S \cdot \theta\varphi_2\varphi_1$ iff $d_2 \in Q(p)$ in $S \cdot \theta\varphi_1\varphi_2$

(51) For any Select operator S and any pointer p, $S \in Q(p)$ in $S \cdot \theta \varphi_2 \varphi_1$ iff

$S \in Q(p)$ in $S \cdot \theta \varphi_1 \varphi_2$ (20)+(30)+(46)+(49)+(50)

(52) For any arc b in P, b is empty in $S \cdot \theta \varphi_2 \varphi_1$ iff b is empty in

$S' \cdot \theta \varphi_2 \varphi_1$ or b is an output arc of a Select operator S and there

is a pointer p such that $S \in Q(p)$ in $S \cdot \theta \varphi_2 \varphi_1$ (2)+Def. 7.1-1

(53) iff b is empty in $S' \cdot \theta \varphi_1 \varphi_2$ or   (4)

(54) b is an output arc of a Select operator S and there is a pointer p

such that $S \in Q(p)$ in $S \cdot \theta \varphi_1 \varphi_2$ (51)

(55) iff b is empty in $S \cdot \theta \varphi_1 \varphi_2$ (2)+Def. 7.1-1

(56) b holds a token of non-pointer value v in $S \cdot \theta \varphi_2 \varphi_1$ iff b holds a

token of value v in $S' \cdot \theta \varphi_2 \varphi_1$ iff b holds a token of value v in

$S' \cdot \theta \varphi_1 \varphi_2$ iff b holds a token of value v in $S \cdot \theta \varphi_1 \varphi_2$ (4)+(2)+Def. 7.1-1

(57) b holds a token of value (p,R) or (p,W) for pointer p in $S \cdot \theta \varphi_2 \varphi_1$

iff b holds a token of value p in $S' \cdot \theta \varphi_2 \varphi_1$ iff b holds a token of

value p in $S' \cdot \theta \varphi_1 \varphi_2$ iff b holds a token of value (p,R) or (p,W) in

$S \cdot \theta \varphi_1 \varphi_2$ (4)+(2)+Def. 7.1-1

(58) For each actor d, there are the same number of firings of d in

$\theta \varphi_2 \varphi_1$ and $\theta \varphi_1 \varphi_2$, and if d is a gate, the kth firings of d in

$\theta \varphi_2 \varphi_1$ and $\theta \varphi_1 \varphi_2$ remove the same control token   (7)

(59) For any actors d and d', the kth firings of d in $\theta \varphi_2 \varphi_1$ and $\theta \varphi_1 \varphi_2$

are preceded by different numbers of firings of d' $\Rightarrow$ d and d' are

d_1 and d_2, and the kth firing of d is either φ_1 or φ_2 $\Rightarrow$ that

firing does not remove a token from an output arc of d'   (8)+(10)

(60) For any arc b which holds pointer-valued tokens in $S \cdot \theta \varphi_2 \varphi_1$ and

$S \cdot \theta \varphi_1 \varphi_2$, either both are read pointers or both are write pointers

         (58)+(59)+Lemma 7.2-5

(61) The heaps in $S \cdot \theta \varphi_2 \varphi_1$ and $S \cdot \theta \varphi_1 \varphi_2$ are identical   (31)+(33)+(4)

(62) $S \cdot \theta \varphi_2 \varphi_1$ and $S \cdot \theta \varphi_1 \varphi_2$ are identical (51)+(52)+(55)+(56)+(57)+(60)+(61)

$\triangle$

Lemma 7.2-6 Given any L_D program P, let Ω be any halted firing sequence starting in any initial modified state S for P. Let $\theta \varphi_2 \varphi_1$ be any prefix of Ω and let Ξ be such that $\Omega = \theta \varphi_2 \varphi_1 \Xi$. If $\theta \varphi_1 \varphi_2$ is a firing sequence starting in S and $S \cdot \theta \varphi_1 \varphi_2$ is identical to $S \cdot \theta \varphi_2 \varphi_1$, then $\Omega' = \theta \varphi_1 \varphi_2 \Xi$ is a halted firing seqeuence starting in S and $\eta(S, \Omega')$ contains the same set of entries as $\eta(S, \Omega)$.

Proof: (The lengthy proof of this intuitive result is in Appendix E.)

$\triangle$

Theorem 7.2-3 Every expansion (Int, J) from $EE(L_D, M)$ satisfies the Commutativity Axiom.

Proof: (All initiations and reconstructions are with respect to Int.)

 (1) Let $\alpha g f \delta$ be any computation in any job $J \in J$ such that $\alpha f g \delta$ is also in J.

 (2) (Int, J) is the expansion of some L_D program P, Int = Int(P), and there is an equivalence class E of initial modified states for P such that $J = J_E$ (1)+Defs. 4.3-1+4.3-3

 (3) Let t be any transfer in $ET_J(\alpha g f \delta)$. Then there is an entry h with T(h) = t such that $\alpha g f \delta h$ is in J   Def. 6.2-2

 (4) There is an initial modified state $S_1 \in E$ for P and a halted firing sequence Ω_1 starting in S_1 such that $\alpha g f \delta h$ is a prefix of some β in J_{S_1, Ω_1} (2)+(3)+Def. 4.3-3

 (5) There is an initial modified state $S_2 \in E$ for P and a halted firing sequence Ω_2 starting in S_2 such that $\alpha f g \delta$ is a prefix of some β'

in J_{S_2, Ω_2} (1)+(2)+Def. 4.3-3

(6) β is a causal permutation of $\eta(S_1, \Omega_1)$. Write it as $\beta = \alpha g f \delta h \gamma$

                                        (4)+Def. 4.3-5

(7) β' is also causal   (5)+Def. 4.3-5

(8) Let εk be any prefix of $\alpha f g \delta h \gamma$. Let e be the execution of which k

is an output entry. $k \in \alpha f g \delta \Rightarrow \varepsilon k$ is a prefix of $\alpha f g \delta \Rightarrow \varepsilon k$ is a

prefix of β' (5)

(9) $\Rightarrow e$ is initiated in ε (7)+Def. 4.2-7

(10) $k \in h \gamma \Rightarrow \exists \iota: \varepsilon k = \alpha f g \delta \iota k \Rightarrow \alpha g f \delta \iota k$ is a prefix of β (6)

(11) $\Rightarrow$ the initiating entry of e is in $\alpha g f \delta \iota$ (8)+(6)+Def. 4.2-7

(12) $\Rightarrow e$ is initiated in $\alpha f g \delta \iota = \varepsilon$ Def. 4.2-6

(13) e is initiated in ε (8)+(9)+(10)+(12)

(14) $\alpha f g \delta h \gamma$ is causal   (8)+(13)+Def. 4.2-7

(15) S_2 and S_1 are equal states   (4)+(5)+(2)

There are now two cases to consider: f and g either are or are not the

initiating entries of two distinct executions in $\alpha g f \delta$.

Case I: f and g are not the initiating entries in $\alpha g f \delta$ of two distinct

executions of actors in P.

(16) $\alpha f g \delta h \gamma$ is a causal permutation of β, hence of $\eta(S_1, \Omega_1)$ (6)+(14)

(17) f and g are input entries of the same execution $Ex(d,k) \Rightarrow f$ is the

initiating entry in $\alpha g f$ iff g is the initiating entry in $\alpha f g$ and

d is in St-DL   Defs. 4.2-6+4.3-2+4.3-1

(18) $\Rightarrow \Phi(\alpha g) = \Phi(\alpha) = \Phi(\alpha f) \Rightarrow [f$ is the initiating entry in $\alpha g f \Rightarrow$

$\Phi(\alpha g f) = \Phi(\alpha)\varphi$, where φ is a firing of d, and $\Phi(\alpha f g) = \Phi(\alpha)\varphi] \wedge$

$[f$ is not the initiating entry in $\alpha g f \Rightarrow \Phi(\alpha g f) = \Phi(\alpha) = \Phi(\alpha f g)]$

$\Rightarrow \Phi(\alpha gf) = \Phi(\alpha fg)$ 　　　　　　　　　　Defs. 4.3-4+4.2-6

(19) f and g are input entries to distinct executions $Ex(d_1,k_1)$ and

$Ex(d_2,k_2)$ respectively $\Rightarrow$ [f is an initiating entry and $d_1 \in$ St-DL

$\Rightarrow$ g is not an initiating entry or $d_2 \notin$ St-DL $\Rightarrow \Phi(\alpha gf) = \Phi(\alpha g)\varphi_1$,

where φ_1 is a firing of d_1, and $\Phi(\alpha g) = \Phi(\alpha) \wedge$

$\Phi(\alpha fg) = \Phi(\alpha f) = \Phi(\alpha)\varphi_1]$ $\wedge$ [g is an initiating entry and $d_2 \in$ St-DL

$\Rightarrow$ f is not an initiating entry or $d_1 \notin$ St-DL $\Rightarrow \Phi(\alpha gf) = \Phi(\alpha)\varphi_2$,

where φ_2 is a firing of d_2 and $\Phi(\alpha fg) = \Phi(\alpha f)\varphi_2 = \Phi(\alpha)\varphi_2] \Rightarrow$

$\Phi(\alpha gf) = \Phi(\alpha fg)$ 　　　　　　　　　　Def. 4.3-4

(20) $\wedge$ [f is not an initiating entry or $d_1 \notin$ St-DL $\wedge$ g is not an initiating

entry or $d_2 \notin$ St-DL $\Rightarrow \Phi(\alpha gf) = \Phi(\alpha) = \Phi(\alpha fg)$] 　　Def. 4.3-4

(21) $\Phi(\alpha gf) = \Phi(\alpha fg)$ 　　　　　　　　　(17)+(18)+(19)+(20)

(22) $\Phi(\alpha gf\delta h\gamma) = \Phi(\beta)$ is the reduction of Ω_1 　　　(6)+(4)+Def. 4.3-5

(23) Let θ be any firing sequence whose reduction is $\Phi(\alpha gf) = \Phi(\alpha fg)$.

For any prefix ι of $\delta h\gamma$, let Δ be such that $\Phi(\alpha gf\iota)$ $(\Phi(\alpha fg\iota))$ is

the reduction of $\theta\Delta$. Then $\Phi(\alpha fg\iota)$ $(\Phi(\alpha gf\iota))$ is also the reduction

of $\theta\Delta$ 　　　　　　　　　　　　　　(21)+Lemma 7.2-4

(24) $\Phi(\alpha fg\delta h\gamma)$ is the reduction of Ω_1 　　　　(22)+(23)+Def. 2.4-5

(25) Let ε'k be any prefix of $\alpha fg\delta h\gamma$. Let Δ be the prefix of Ω_1 whose

reduction is $\Phi(\varepsilon')$. $k \in \alpha fg\delta \Rightarrow \varepsilon'$k is a prefix of $\alpha fg\delta \Rightarrow \varepsilon'$k is a

prefix of $\beta' \Rightarrow$ letting Δ_2 be the prefix of Ω_2 whose reduction is

$\Phi(\varepsilon')$, Δ is a firing sequence starting in S_1 which is equal to Δ_2

　　　　　　　　　　　(5)+Lemma 7.2-2+Defs. 2.3-1+2.4-5

(26) $\Rightarrow S_2 \cdot \Delta_2$ equals $S_1 \cdot \Delta$ 　　　　　　　(15)+Thm. 7.1-2

(27) $k \notin \alpha fg\delta \Rightarrow \exists \iota: \varepsilon'$k $= \alpha fg\iota k \Rightarrow |\Phi(\alpha fg)| \leq |\Phi(\varepsilon')| \Rightarrow \Delta$ can be written as

θX where the reduction of θ is $\Phi(\alpha fg)$ 　　　Defs. 4.3-4+2.4-5

(28) ⇒ $\Phi(\alpha gf\iota)$ is the reduction of $\theta X = \Delta$ (23)

(29) ⇒ there is a prefix ϵk of $\alpha gf\delta h\gamma$ such that, for Δ_1 the prefix of Ω_1

      whose reduction is $\Phi(\epsilon) = \Phi(\alpha gf\iota)$, Δ is identical, hence equal,

      to Δ_1 (25)+Def. 2.4-5

(30) ⇒ $S_1 \cdot \Delta_1$ equals $S_1 \cdot \Delta$ (15)+Thm. 7.1-2

(31) $\alpha fg\delta h\gamma$ is in J_{S_1, Ω_1} (4)+(5)+(16)+(24)+(25)+(26)+(27)+(30)+Lemma 7.2-8

(32) $\alpha fg\delta h$ is in $J_E = J$ (31)+(4)+(5)+Def. 4.3-3

(33) $T(h)$ is in $ET_J(\alpha fg\delta)$ (32)+Def. 6.2-2

Case II: f and g are the initiating entries of two distinct executions

$e_1 = Ex(d_1, k_1)$ and $e_2 = Ex(d_2, k_2)$, where d_1 and d_2 are both labels of

actors in P.

(34) αgf and αfg are both in J   (4)+(5)+(2)+Def. 4.3-3

(35) There are prefixes $\theta\varphi_2\varphi_1$ of Ω_1 and $\theta'\varphi_1'\varphi_2'$ of Ω_2, whose reductions

      are $\Phi(\alpha gf)$ and $\Phi(\alpha fg)$, such that θ' equals θ and, for i=1,2, φ_i

      (φ_i') is the k_i^{th} firing of d_i. Furthermore, φ_1 and φ_2 potentially

      interfere in $\theta\varphi_2\varphi_1$ iff $Ent(e_1, 1)$ and $Ent(e_2, 1)$ are in the same

      access history and $e_1 \in R(e_2)$ in αgf (2)+(34)+(4)+(5)+Lemma 7.2-3

(36) $Ent(e_1, 1)$ and $Ent(e_2, 1)$ are in the same access history and $e_1 \in R(e_2)$

      in αgf ⇒ (Int, J) does not satisfy the Freedom-from-conflict Axiom

                   (1)+(34)+Axiom 6.2-7

(37) φ_1 and φ_2 do not potentially interfere in $\theta\varphi_2\varphi_1$ (35)+(36)+Thm. 7.2-1

(38) $\theta\varphi_1\varphi_2$ equals $\theta'\varphi_1'\varphi_2'$ (35)+Def. 2.4-5

(39) Let ω be $\eta(S_1, \theta\varphi_2\varphi_1)$, and let NAR be the node activation record

      derived from $\theta\varphi_2\varphi_1$ and ω. For any pointer-node pair (p,n), (p,n)

      is in a Copy firing in $\theta\varphi_2\varphi_1$ iff there is a Copy label C and an

      integer k such that there are at least k firings of C in $\theta\varphi_2\varphi_1$

and the kth is $(C,(p,n))$ iff $Ex(C,k)$ is initiated in ω and the kth

firing of C in Ω_1 is $(C,(p,n))$ Lemma 4.3-1+Defs. 4.3-1+4.2-6

(40) iff (p,n) is in ran NAR   Def. 5.2-4

(41) The multiset of pointer-node pairs in the Copy firings in $\theta\varphi_2\varphi_1$ is

ran NAR, which is consistent with the heap in S_1

$$(39)+(40)+\text{Lemma } 5.2\text{-}2$$

(42) $\theta\varphi_1\varphi_2$ is a firing sequence starting in S_1

$$(35)+(15)+(38)+(41)+\text{Cor. } 7.1\text{-}2$$

(43) $S_1\cdot\theta\varphi_1\varphi_2$ and $S_1\cdot\theta\varphi_2\varphi_1$ are identical states $(35)+(42)+(37)+\text{Thm. } 7.2\text{-}2$

(44) Let Ξ be such that $\Omega_1 = \theta\varphi_2\varphi_1\Xi$ and let Ω_1' be $\theta\varphi_1\varphi_2\Xi$. Then Ω_1' is a

halted firing sequence starting in S_1, and $\eta(S_1,\Omega_1')$ consists of

the same set of entries as $\eta(S_1,\Omega_1)$ $(4)+(35)+(42)+(43)+\text{Lemma } 7.2\text{-}6$

(45) $\alpha f g \delta h \gamma$ is a permutation of $\eta(S_1,\Omega_1)$ (6)

(46) $\alpha f g \delta h \gamma$ is a permutation of $\eta(S_1,\Omega_1')$ (45)+(44)

(47) $\Phi(\alpha g f \delta h \gamma)$ is the reduction of $\Omega_1 = \theta\varphi_2\varphi_1\Xi$ (22)+(44)

(48) $\Phi(\alpha f g)$ is the reduction of $\theta\varphi_1\varphi_2$ (35)+Def. 2.4-5

(49) $\Phi(\alpha f g \delta h \gamma)$ is the reduction of $\theta\varphi_1\varphi_2\Xi = \Omega_1'$

$$(35)+(48)+(47)+(44)+\text{Lemma } 7.2\text{-}4$$

(50) Let $\varepsilon'k$ be any prefix of $\alpha f g \delta h \gamma$. Let Δ be the prefix of Ω_1' whose

reduction is $\Phi(\varepsilon')$. $k\in\alpha f g \delta \Rightarrow \varepsilon'k$ is a prefix of $\alpha f g \delta \Rightarrow$ letting Δ_2

be the prefix of Ω_2 whose reduction is $\Phi(\varepsilon')$, Δ is a firing sequence

starting in S_1 which is equal to Δ_2 (44)+Defs. 2.3-1+2.4-5

(51) $\Rightarrow S_2\cdot\Delta_2$ equals $S_1\cdot\Delta$ (15)+Thm. 7.1-2

(52) $S_1\cdot\theta\varphi_1\varphi_2$ equals $S_1\cdot\theta\varphi_2\varphi_1$ (43)+Defs. 7.1-2+3.4-1

(53) $k\notin\alpha f g \delta \Rightarrow \exists\iota: \varepsilon'k = \alpha f g \iota k \Rightarrow |\Phi(\alpha f g)| \le |\Phi(\varepsilon')| \Rightarrow \Delta$, whose reduction

is $\Phi(\alpha f g \iota)$, can be written as $\Delta = \theta\varphi_1\varphi_2\chi$ (45)+(48)+(50)+Def. 4.1-4

1.0	2.8	2.5
1.1	3.2	2.2
	3.6	
	4.0	2.0
		1.8
1.25	1.4	1.6
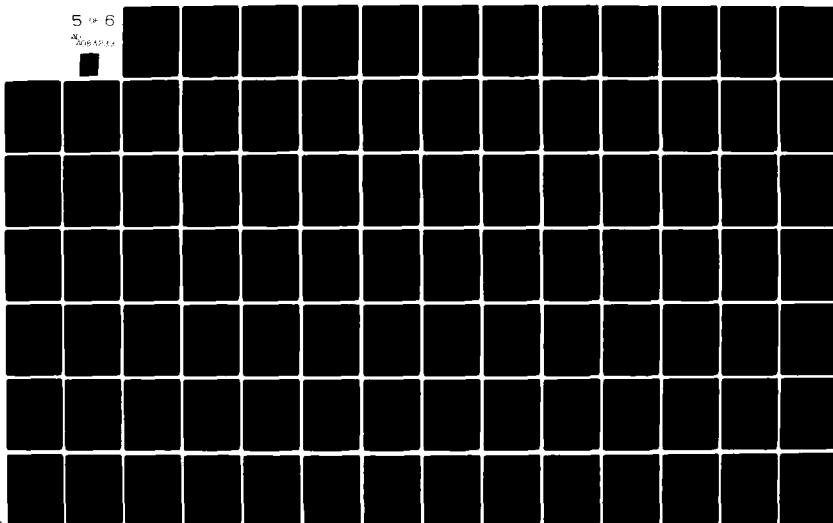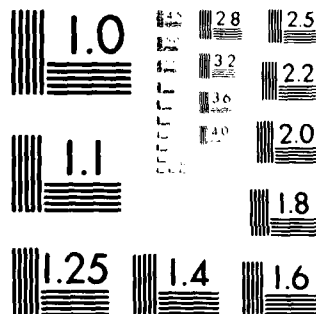
MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS 1963 A

(54) $\Rightarrow \Phi(\alpha gf\iota)$ is the reduction of $\theta\varphi_2\varphi_1 X$ (35)+(48)+Lemma 7.2-4

(55) $\Rightarrow$ there is a prefix $\epsilon k = \alpha gf\iota k$ of $\alpha gf\delta h\gamma$ such that, for Δ_1 the

prefix of Ω_1 whose reduction is $\Phi(\epsilon) = \Phi(\alpha gf\iota)$, $S_1 \cdot \Delta_1 = S_1 \cdot \theta\varphi_2\varphi_1 X$

equals $S_1 \cdot \Delta = S_1 \cdot \theta\varphi_1\varphi_2 X$ (50)+(44)+(52)+Lemma 7.2-2+Thm. 7.1-2

(56) $\alpha fg\delta h\gamma$ is in $J_{S_1,\Omega_1'}$

     (4)+(5)+(44)+(14)+(46)+(49)+(50)+(51)+(53)+(55)+Lemma 7.2-8

(57) $\alpha fg\delta h$ is in $J_E = J$ (56)+(4)+(5)+Def. 4.3-3

(58) $T(h)$ is in $ET_J(\alpha fg\delta)$ (57)+Def. 6.2-2

In either case then,

(59) For any transfer t, $t \in ET_J(\alpha gf\delta) \Rightarrow t \in ET_J(\alpha fg\delta)$ (3)+(35)+(58)

By symmetry,

(60) For any transfer t, $t \in ET_J(\alpha fg\delta) \Rightarrow t \in ET_J(\alpha gf\delta)$;

i.e., $ET_J(\alpha fg\delta) = ET_J(\alpha gf\delta)$

$\triangle$

7.2.4 Persistence

The Persistence Axiom asserts that for any job J and any computation αg in J, for any transfer $t \neq T(g)$, $t \in ET_J(\alpha) \Rightarrow t \in ET_J(\alpha g)$. In other words, once a transfer becomes eligible, it remains so until the appearance of an entry having that transfer. The analogous property of the schema model of data flow — that once enabled, an actor remains enabled (with the same tokens on its input arcs) until it fires — is easily demonstrated:

Theorem 7.2-4 Let S be any initial modified state for any program P. Let $\theta\varphi$ be any firing sequence starting in S, and let d be the actor of which the last firing φ is a firing. Then for any actor d' $\neq$ d, each input arc

of d' which holds a token in $S \cdot \theta$ holds the same token in $S \cdot \theta \varphi$, and d' is enabled in $S \cdot \theta \Rightarrow$ d' is enabled in $S \cdot \theta \varphi$.

Proof:

(1) If d is a Select, its output arcs are all empty in $S \cdot \theta$

Defs. 3.3-6+2.1-4

(2) Any arc which has a token in $S \cdot \theta$ but no token in $S \cdot \theta \varphi$ is an input arc of d (1)+Defs. 3.3-9+2.1-5

(3) Let d' be any actor except d. No input arc of d is an input arc of d' and no output arc of d is an output arc of d' Def. 2.1-1

(4) Any arc which holds a token in $S \cdot \theta \varphi$ which it does not hold in $S \cdot \theta$ is either an output arc of d or a data-output arc of a Select operator S for which there is a p such that $S \in Q(p)$ in $S \cdot \theta$

Defs. 3.3-9+2.1-5

(5) No arc holding a token in $S \cdot \theta$ is an output arc of d Defs. 3.3-6+2.1-4

(6) No arc which has a token in $S \cdot \theta$ is a data-output arc of a Select operator S for which there is a p such that $S \in Q(p)$ in $S \cdot \theta$ Cor. 7.1-1

(7) Each input arc of d' which holds a token in $S \cdot \theta$ holds the same token in $S \cdot \theta \varphi$ (2)+(3)+(4)+(5)+(6)

(8) d' is enabled in $S \cdot \theta \Rightarrow$ there is no pointer p such that $d' \in Q(p)$ in $S \cdot \theta$ (3)+Def. 3.3-6

(9) $\Rightarrow$ since φ is not a firing of d', there is no pointer p such that $d' \in Q(p)$ in $S \cdot \theta \varphi$ Def. 3.3-9

(10) $\Rightarrow$ [d' is not enabled in $S \cdot \theta \varphi \Rightarrow$ either there is an input arc of d' which holds a token in $S \cdot \theta$ but holds either no token or a different control token in $S \cdot \theta \varphi$, or there is an output arc of d' which has no token in $S \cdot \theta$ but has one in $S \cdot \theta \varphi$] Def. 3.3-6

(11) $\wedge$ no output arc of d' is an output arc of a Select operator S for

which there is a p such that $S \in Q(p)$ in $S \cdot \theta$ (8)+Def. 2.1-1

(12) $\Rightarrow$ d' is enabled in $S \cdot \theta \varphi$ (2)+(3)+(4)+(5)+(6)

$\triangle$

Proving persistence in the entry-execution model is considerably more difficult: Any computation αg in a job J is a prefix of some $\beta \in J_{S,\Omega}$ where S is an initial modified state and Ω is a halted firing sequence starting in S. For any $t \in ET_J(\alpha)$, there is a computation $\alpha h \in J$ with $T(h) = t$, so there is an initial modified state S' equal to S, and a halted firing sequence Ω' starting in S', such that αh is a prefix of some $\beta' \in J_{S',\Omega'}$. The goal is to construct a halted firing sequence X starting in S such that there is a computation $\alpha g \bar{h} \delta$ in $J_{S,X}$, where $T(\bar{h}) = t$.

Every entry in αg must be in $\eta(S,\Omega)$. The non-dummy targets of those entries can be partitioned into two sets of executions: $AF = \{Ex(d,k) \mid d \notin DL$ and $Ex(d,k)$ is initiated in $\alpha\}$ and $EF = \{Ex(d,k) \mid d \notin DL$ and $Ex(d,k)$ has an input entry in αg but is not initiated in $\alpha\}$. For any $e = Ex(d,k)$, $e \in AF$ iff there are k firings of d in the prefix θ of Ω whose reduction is $\Phi(\alpha)$ (Theorem 4.3-2) iff all input entries to e in α are in $\eta(S,\theta)$ iff for any X having θ as a prefix, all input entries to e in α are in $\eta(S,X)$. It is much harder to accomodate an input entry f to an execution $Ex(d,k)$ in EF: X must be constructed so that there is a kth firing of d in X (which there is not in θ) and that firing removes the same tokens from the same set of input arcs as the kth firing of d in Ω. Similarly, the target $Ex(d',k')$ of h is not initiated in α, so there are not k' firings of d' in θ; nonetheless, there must also be a k'^{th} firing of d' in X which removes the same tokens from the same set of input arcs as the k'^{th} firing of d' in Ω'.

For any input entry f in αg of an execution $e = Ex(d,k) \in EF$, there is

a j such that T(f) has destination Dst(e,j), and there is a δ such that δf

is a prefix of β. Since f is in $\eta(S,\Omega)$, there is a prefix $\Delta\phi$ of Ω in which

ϕ is the kth firing of d. Since there are fewer than k firings of d in θ,

θ is a prefix of Δ. In $S \cdot \Delta$, d is enabled, and if d is a merge gate and

its number-j input arc b is its T (F) input arc, then d's control input

arc holds a true (false) token. The only reasonable way to guarantee that

a X can be constructed, having θ as a prefix and containing a kth firing

of d which is not in θ, is to guarantee that d is enabled in $S \cdot \theta$; further-

more, if d is a merge gate and b is its T (F) input arc, then d's control

input arc should hold a true (false) token in $S \cdot \theta$. Then by Theorem 7.2-4,

any X with θ as a prefix must contain a kth firing of d which removes a

token from b. The "worst case" of f = g, so $\delta = \alpha$, motivates one of the

unexplained requirements included in Definition 4.3-5, that for Ξ the

prefix of Ω whose reduction is $\Phi(\delta)$, d is enabled in $S \cdot \Xi$, with a prescribed

control input if it is a merge gate.

If f ≠ g, then Ξ may be a prefix of θ; i.e.,

$$|\Xi| \leq |\theta| \leq |\Delta|$$

There can be no firing of d in Δ which is not in Ξ, as the following

reasoning shows: The token on b in $S \cdot \Xi$ remains there until the next firing

of d (Theorem 7.2-4). If b is an output arc of an actor c, then f is an

output entry of Ex(c,n) where there are n firings of c in Δ. By causality,

Ex(c,n) is initiated before f; i.e., in δ, so by Theorem 4.3-2, there are

at least n firings of c in Ξ. Therefore, every firing of c in Δ is in Ξ.

If there are any firings of d in Δ but not in Ξ, the first of them neces-

sarily removes from b the token which is on it in $S \cdot \Xi$. But there is a

token on b in $S \cdot \Delta$, which could only be placed there by a firing of c in Δ which is not in Ξ. Thus, there are the same number of firings of d (k-1) in Δ and Ξ, hence in θ. Furthermore, d is enabled in $S \cdot \theta$, with the proper control input if it is a merge gate, and d's input arcs hold the same tokens in $S \cdot \Delta$ and in $S \cdot \Xi$, hence in $S \cdot \theta$. Finally, if b is an output arc of actor c, then there are the same number of firings of c in Δ and Ξ, hence in θ (Lemma 7.2-7 below).

The above is true for any $Ex(d,k) \in EF$, so every actor in the set $\{d \mid \exists k: Ex(d,k) \in EF\}$ is enabled in $S \cdot \theta$. Similarly, for the target $Ex(d',k')$ of h, d' is enabled in $S' \cdot \theta'$, where θ' is the prefix of Ω' whose reduction is $\Phi(\alpha)$. Since the reductions of θ and θ' are both $\Phi(\alpha)$, θ and θ' are equal; since S and S' are equal states, $S' \cdot \theta'$ equals $S \cdot \theta$. Therefore, d' is also enabled in $S \cdot \theta$ (Corollary 7.1-2), with the same control input if it is a gate. I.e., all actors in the set $EA = \{d \mid \exists k: Ex(d,k) \in EF\} \cup \{d'\}$ are enabled in $S \cdot \theta$. As will be seen, any ordering $d_1, d_2, \ldots, d_m$ of the actors in EA is an acceptable firing order in χ if it satisfies the following: If g initiates execution $Ex(d'',k'')$, then $d_1 = d''$ and $d_2 = d'$, where $Ex(d',k')$ is h's target; otherwise, $d_1 = d'$. No firing of an actor in EA can disable or affect the input tokens of any other enabled actor in the set (Theorem 7.2-4). Therefore, $\theta \varphi_1 \varphi_2 \ldots \varphi_m$ where φ_i is a firing of d_i, is a firing sequence starting in S in which φ_i removes the tokens which are on d_i's input arcs in $S \cdot \theta$ (Corollary 7.2-1 below). Now χ is chosen to be any halted firing sequence starting in S having $\theta \varphi_1 \varphi_2 \ldots \varphi_m$ as a prefix.

For any entry $f \in \alpha g$, with transfer $(s, Dst(e,j))$ where $e = Ex(d,k)$, there are two cases to consider:

Case I: d∉DL

Then e∈AF∪EF. If e is in AF, it has already been argued that f is in $\eta(S,\theta)$, hence in $\eta(S,X)$. Otherwise, d is in EA, so there is some i such that φ_i is a firing of d_i in $\theta\varphi_1\varphi_2\ldots\varphi_m$. Since there are exactly k-1 firings of d in θ (Lemma 7.2-7), φ_i is the kth firing of d in X. As above, there is a prefix $\Delta\varphi$ of Ω in which φ is the kth firing of d, φ removes a token of value V(f) from d's number-j input arc b, and s is Source(b,S,Δ). If d is a merge gate and b is its T (F) input arc, then d's control input arc holds a <u>true</u> (<u>false</u>) token in $S\cdot\theta$ (Lemma 7.2-7), hence in $S\cdot\theta\varphi_1\ldots\varphi_{i-1}$ (Corollary 7.2-1). The token on b in $S\cdot\Delta$ is on b in $S\cdot\theta$, and so is on b in $S\cdot\theta\varphi_1\ldots\varphi_{i-1}$. Because of that token, b is an output arc of actor c ⇒ c is not enabled in $S\cdot\theta$ ⇒ c is not in EA ⇒ the number of firings of c in $\theta\varphi_1\ldots\varphi_{i-1}$ equals the number in θ, which equals the number in Δ (Lemma 7.2-7); from Lemma 7.1-2, then, Source(b,S,Δ), which is s, equals Source(b,S,$\theta\varphi_1\ldots\varphi_{i-1}$). Therefore, the kth firing of d in X, φ_i, removes a token of value V(f) from d's number-j input arc, so there is an entry with value V(f) and transfer (s,Dst(Ex(d,k),j)) in $\eta(S,X)$; i.e., f is in $\eta(S,X)$.

Case II: d∈DL

Either d = ("OD",n) or d = (c,n) for some actor label c. There is a token of value V(f) on arc b in $S\cdot\Omega$, where if d = ("OD",n), b is the number-n program output arc, and otherwise b is the number-n input arc of c; furthermore, s = Source(b,S,Ω). There is a δ such that δf is a prefix of αg, and there is a prefix Ξ of Ω whose reduction is $\Phi(\delta)$. As before, Ξ is a prefix of θ, hence of X. By the second unexplained requirement in Definition 4.3-5, there is a token on b in $S\cdot\Xi$, and if b is an input arc

of c, no firing sequence starting in $S \cdot \Xi$ contains a firing of c. If b

is a program output arc, then the token on b in $S \cdot \Xi$ is still on b in both

$S \cdot \Omega$ and $S \cdot \chi$. If b is an input arc of c, then c never fires after Ξ in

either Ω or χ, so the token on b in $S \cdot \Xi$ is there in $S \cdot \Omega$ and $S \cdot \chi$. Further-

more, if b is an output arc of an actor c', that token keeps c' disabled,

so there are the same number of firings of c' in both Ω and χ as there are

in Ξ; hence, Source(b,S,χ) = Source(b,S,Ω) = s. Therefore, f is in $\eta(S,\chi)$.

Thus it is seen that every entry in αg is in $\eta(S,\chi)$. Very similar

reasoning, together with the fact that $S' \cdot \theta'$ equals $S \cdot \theta$, proves that there

is an entry $\bar{h}$ in $\eta(S,\chi)$ with at least the same transfer as h. Letting δ

be $\eta(S,\chi)$ with every entry in $\alpha g \bar{h}$ stricken out — i.e., entry f_2 follows

entry f_1 in δ iff f_2 follows f_1 in $\eta(S,\chi)$ and neither is in $\alpha g \bar{h}$ — $\alpha g \bar{h} \delta$ is

a permutation of $\eta(S,\chi)$. If $\alpha g \bar{h} \delta$ is in $J_{S,\chi}$, then $\alpha g \bar{h}$ is in J, so

$T(\bar{h})$ = $T(h)$ = t is in $ET_J(\alpha g)$, as was to be shown.

It is easily seen that $\alpha g \bar{h} \delta$ is causal: For any prefix γf of $\alpha g \bar{h} \delta$,

in which f is an output entry of an execution e, if f is in αg, or is $\bar{h}$,

then e is initiated in γ because αg and αh are causal. If f is in δ,

then the initiating entry of e precedes f, either because it is in $\alpha g \bar{h}$ or

because it precedes f in the causal $\eta(S,\chi)$.

Proving that $\Phi(\alpha g \bar{h} \delta)$ is the reduction of χ is more difficult. Let-

ting the targets of g and $\bar{h}$ be Ex(d'',k'') and Ex(d',κ') respectively,

$\Phi(\alpha g \bar{h})$ is given by:

   a. if neither g nor $\bar{h}$ is an initiating entry, then $\Phi(\alpha)$, the reduction

       of θ

   b. if only g is an initiating entry, then $\Phi(\alpha)\varphi_a$, where φ_a is a firing

       of d''

c. if only $\bar{h}$ is an initiating entry, then $\Phi(\alpha)_{\varphi_b}$, where φ_b is a firing of d'

d. if both are initiating entries, then $\Phi(\alpha)_{\varphi_a \varphi_b}$.

By construction, if g is an initiating entry, then there is a prefix $\theta\varphi_1\varphi_2$ of X in which φ_1 is a firing of d" and φ_2 is a firing of d'; otherwise, there is a prefix $\theta\varphi_1$ in which φ_1 is a firing of d'. From these two observations, $\Phi(\alpha g\bar{h})$ is the reduction of a prefix Λ of X, the length of which, m, equals the length of $\Phi(\alpha g\bar{h})$.

Since $\Phi(\eta(S,X))$ is the reduction of X (Lemma 4.3-3), the nth firing in X is a firing of actor d iff the nth execution initiated in $\eta(S,X)$ is an execution of d. For $n \leq m$, this is iff the nth firing in $\Phi(\alpha g\bar{h}\delta)$ is a firing of d. The first m executions initiated in $\eta(S,X)$ are initiated in $\eta(S,\Lambda)$. Ex(d,k) is among these iff there are at least k firings of d in Λ iff Ex(d,k) is initiated in $\alpha g\bar{h}$ (by Theorem 4.3-2, since $\Phi(\alpha g\bar{h})$ is the reduction of Λ). Therefore, for any $n > m$ and any i, the ith initiating entry in $\alpha g\bar{h}\delta$, f', precedes the nth such entry, f, iff f is in δ and either $i \leq m$, or $i > m$ and f' is in δ iff the execution initiated by f' is initiated among the first m in $\eta(S,X)$, or f' precedes f in $\eta(S,X)$ (by construction of δ) iff the the ith initiating entry in $\eta(S,X)$ (which is not necessarily f') precedes the nth initiating entry (which is necessarily f). Thus, the nth execution initiated in $\eta(S,X)$ is an execution of d iff the nth execution initiated in $\alpha g\bar{h}\delta$ is an execution of f; hence the nth firing in the reduction of X is a firing of d iff the nth firing in $\Phi(\alpha g\bar{h}\delta)$ is a firing of d. I.e., $\Phi(\alpha g\bar{h}\delta)$ is the reduction of X.

The remainder of the proof that $\alpha g\bar{h}\delta$ is in $J_{S,X}$ uses reasoning developed earlier in conjunction with the proof of commutativity. It has been

seen that $\alpha g \bar{h} \delta$ is a causal permutation of $\eta(S,X)$ for which $\Phi(\alpha g \bar{h} \delta)$ is the reduction of X. The goal then is to show that for every prefix γf of $\alpha g \bar{h} \delta$, letting Δ be the prefix of X whose reduction is $\Phi(\gamma)$, there is an initial modified state S_1 equal to S and a halted firing sequence Ω_1 starting in S_1, and there is a prefix $\gamma' f'$ of some $\varepsilon \in J_{S_1, \Omega_1}$, where $T(f') = T(f)$ such that, letting Δ' be the prefix of Ω_1 whose reduction is $\Phi(\gamma')$, $S_1 \cdot \Delta'$ equals $S \cdot \Delta$. If f is in αg, then since αg is a prefix of $\beta \in J_{S, \Omega}$, $S_1 = S$, $\Omega_1 = \Omega$, $\gamma' f' = \gamma f$, and $\Delta' = \Delta$; clearly $S_1 \cdot \Delta'$ equals $S \cdot \Delta$. If f is in δ, then there is a γ' such that $\gamma' f$ is a prefix of $\eta(S,X) \in J_{S,X}$. All input entries to f's target execution e are consecutive in $\eta(S,X)$, so all input entries to e in δ are consecutive. For any other execution $e' \neq e$, the initiating entry of e' is not between f and e's initiating entry in $\alpha g \bar{h} \delta$, and so it precedes f (is in $\dot{\gamma}$) iff it precedes e's initiating entry in $\alpha g \bar{h} \delta$ iff it precedes e's initiating entry in $\eta(S,X)$ (by the above paragraph) iff it precedes f in $\eta(S,X)$ (is in γ'). Therefore, there are the same number of initiating entries in γ and γ', so there is some n such that $|\Phi(\gamma')| = |\Phi(\gamma)| = n$. $\Phi(\gamma)$ is the length-n prefix of $\Phi(\alpha g \bar{h} \delta)$ and $\Phi(\gamma')$ is the length-n prefix of $\Phi(\eta(S,X))$; both $\Phi(\alpha g \bar{h} \delta)$ and $\Phi(\eta(S,X))$ are the reduction of X, so $\Phi(\gamma') = \Phi(\gamma)$. I.e., γ' and γ have the same reduction, so $S_1 \cdot \Delta'$ equals $S \cdot \Delta$.

Finally, if $f = \bar{h}$, f's target is $Ex(d',k')$. Since that execution is not initiated in $\gamma = \alpha g$, if g initiates an execution, it is not an execution of d'. Therefore, $\Phi(\gamma)$, the reduction of Δ, is either $\Phi(\alpha)$ or $\Phi(\alpha)\varphi$ where φ is not a firing of d'. There is a prefix αh of $\beta' \in J_{S', \Omega'}$ such that $T(h) = T(\bar{h}) = T(f)$, and the reduction of Δ is either $\Phi(\alpha)$ or $\Phi(\alpha)\varphi$. Letting Δ' be the prefix of Ω' whose reduction is $\Phi(\alpha)$, Δ' equals either

Δ, or θ where $\Delta = \theta\varphi$ and φ is not a firing of d'. Therefore, $S' \cdot \Delta'$ equals either $S \cdot \Delta$ or $S \cdot \theta$. If $S' \cdot \Delta'$ equals $S \cdot \Delta$, then the reasoning given at the end of the discussion of Case I of the proof of commutativity (Section 7.2.3) applies for all f in $\alpha g \bar{h} \delta$, so $\alpha g \bar{h} \delta$ is in $J_{S,\chi}$. If $S' \cdot \Delta'$ equals $S \cdot \theta$, however, the following extended deduction is needed for $f = \bar{h}$.

Since αh is a prefix of $\beta' \in J_{S',\Omega'}$, $d' \notin DL \Rightarrow d'$ is enabled in $S' \cdot \Delta' \Rightarrow$ by Corollary 7.1-2, d' is enabled in the equal state $S \cdot \theta \Rightarrow$ by Theorem 7.2-4, d' is enabled in $S \cdot \theta\varphi$, since φ is not a firing of d'; furthermore, if d' is a gate, its control input arc holds the same token in $S \cdot \theta$, hence in $S \cdot \theta\varphi$, as in $S' \cdot \Delta'$. Otherwise, $d' \in DL \Rightarrow d' = (c,n)$ and there is a particular arc b (the number-n input arc of the actor labelled c if $c \neq$ "OD") which holds a token in $S' \cdot \Delta'$; furthermore, if $c \neq$ "OD", no firing sequence starting in $S' \cdot \Delta'$ contains a firing of $c \Rightarrow b$ holds a token in $S \cdot \theta$, and no firing sequence starting in $S \cdot \theta$ contains a firing of c (Corollary 7.1-2) $\Rightarrow \varphi$ in particular is not a firing of $c \Rightarrow$ there is a token on b in $S \cdot \theta\varphi = S \cdot \Delta$ (Theorem 7.2-4) and no firing sequence starting in $S \cdot \Delta$ contains a firing of c. Therefore, $\alpha g \bar{h} \delta$ is in $J_{S,\chi}$, so $\alpha g \bar{h}$ is in J and $T(\bar{h}) = T(h) = t$ is in $ET_J(\alpha g)$, as was to be proven. (As noted earlier, for purposes of efficiency, a single lemma, Lemma 7.2-8 below, is fashioned to cover the cases both that $S' \cdot \Delta'$ equals $S \cdot \theta$ and that $S' \cdot \Delta'$ equals $S \cdot \Delta$.)

The rigorous proof that every expansion satisfies the Persistence Axiom is given on the following pages.

Lemma 7.2-7 Given any L_D program P, let Int(P) be (St,$/$,IE). Let S be any initial modified state for P and let Ω be any halted firing sequence starting in S. Let αh be any prefix of any $\beta \in J_{S,\Omega}$, and let θ be the prefix of Ω whose reduction is $\Phi(\alpha)$. Let $e = Ex(d,k)$ be any execution which has an input entry in αh but is not initiated in α, and in which $d \in ST\text{-}DL$. Let $g = Ent(e,j)$ be any input entry to e in αh and let b be d's number-j input arc. Finally, let $\Delta \phi$ be the prefix of Ω in which ϕ is the kth firing of d. Then

A: d is enabled in $S \cdot \theta$, and if d is a merge gate and b is it T (F) input arc, then d's control input arc has a <u>true</u> (<u>false</u>) token in $S \cdot \theta$.

B: There are exactly k-1 firings of d in θ.

C: The token on b in $S \cdot \Delta$ is on b in $S \cdot \theta$.

D: b is an output arc of actor c $\Rightarrow$ there are the same number of firings of c in θ as in Δ.

<u>Proof</u>:

(1) β is a causal permutation of $\eta(S,\Omega)$, so g is in $\eta(S,\Omega)$ Def. 4.3-5

(2) Let T(g) be (s,Dst(e,j)). Then ϕ removes a token from b in Ω, and
      s = Source(b,S,Δ)   (1)+Alg. 4.3-1

(3) Let δ be such that δg is a prefix of β. Let Ξ be the prefix of Ω whose reduction is $\Phi(\delta)$. Then Ξ and Δ are firing sequences starting in S Def. 2.3-1

(4) d is enabled in $S \cdot \Xi$ and if d is a merge gate and b is it T (F) input arc, then d's control input arc has a <u>true</u> (<u>false</u>) token in $S \cdot \Xi$ (3)+Def. 4.3-5

(5) e is not initiated in δ or in α   (3)+(1)+Def. 4.2-6

(6) There are fewer than k firings of d in Ξ and in θ

$\qquad$ (3)+(1)+(5)+Thm. 4.3-2

(7) Ξ is a prefix of Δ, as is θ   (6)

(8) Let s = Src(Ex(c',n),i). c'∈DL ⇒ if b is an output arc of an actor

   c, then there are zero firings of c in Δ, hence in Ξ

$\qquad$ (2)+(3)+Lemma 7.1-3

(9) c'∈St-DL ⇒ c' = c, the label of a actor in P, b is an output arc

   of c, and there are exactly n firings of c in Δ   (8)+(2)+Alg. 4.3-1

(10) ∧ Ex(c,n) is initiated in δ   (3)+(2)+(8)+(1)+Defs. 4.2-5+4.2-7

(11) ⇒ there are at least n firings of c in Ξ   (3)+(1)+Thm. 4.3-2

(12) ⇒ there are exactly n firings of c in Ξ as well as in Δ   (7)+(9)

(13) If b is an output arc of an actor c, then there are the same number

   of firings of c in Ξ as in Δ   (8)+(9)+(12)

(14) There is a token on b in $S \cdot Ξ$ (4)+Defs. 3.3-6+2.1-4

(15) There is no p such that c∈Q(p) in $S \cdot Ξ$ (14)+Cor. 7.1-1

(16) There is a prefix Λ of Ω, $|Ξ| < |Λ| \le |Δ|$ such that, for some p,

   c∈Q(p) in $S \cdot Λ$ ⇒ there is a prefix $Χφ_c$ of Ω, $|Ξ| < |Χφ_c| \le |Δ|$

   such that c∉Q(p) in $S \cdot Χ$ but c∈Q(p) in $S \cdot Χφ_c$ (15)

(17) ⇒ $φ_c$ is a firing of c   Def. 3.3-9

(18) If b is an output arc of an actor c, then there is no prefix Λ of

   Ω, $|Ξ| \le |Λ| \le |Δ|$ such that, for some p, c∈Q(p) in $S \cdot Λ$

$\qquad$ (15)+(16)+(17)+(13)

Prove that A, B, and C are true of every prefix Χ of Δ longer than Ξ.

Proof is by induction on the length of Χ. Induction hypotheses are A with

Χ substituted for θ, and

E: there are the same number of firings of d in X as in Ξ, and

F: if there is a token on b in $S \cdot \Xi$, then the same token is on b in $S \cdot X$.

Basis: $X = \Xi$. E and F are vacuously true.

(19) A   (4)

Induction step: Assume the induction hypotheses are true for any X,

$|\Xi| \leq |X| < |\Delta|$, and consider prefix $X\varphi$ of Ω.

(20) d is enabled in $S \cdot X$ and if d is a merge gate and b is its T (F)

     input arc, then d's control input arc holds a <u>true</u> (<u>false</u>) token

     in $S \cdot X$ ind. hyp.

(21) φ is a firing of d $\Rightarrow$ there is no token on b in $S \cdot X\varphi$

                                    (20)+Defs. 3.3-9+2.2-1+2.1-5

(22) $\Rightarrow$ there is a prefix $\Lambda\varphi'$ of Ω, $|X\varphi| < |\Lambda\varphi'| \leq |\Delta|$ such that there

     is no token on b in $S \cdot \Lambda$ but there is one in $S \cdot \Lambda\varphi'$ (2)

(23) $\Rightarrow$ b is an output arc of an actor c and either φ' is a firing of c

     or there is a p such that $c \in Q(p)$ in $S \cdot \Lambda$ Defs. 3.3-9+2.2-5

(24) φ' is not a firing of c   (22)+(13)

(25) There is no p such that $c \in Q(p)$ in $S \cdot \Lambda$ (22)+(18)

(26) φ is not a firing of d   (21)+(23)+(24)+(25)

(27) A for $X\varphi$ (20)+(26)+Thm. 7.2-4

(28) E for $X\varphi$ (26)+ind. hyp.

(29) If there is a token on b in $S \cdot \Xi$, then the same token is on b in $S \cdot X$

                                          ind. hyp.

(30) F for $X\varphi$ (29)+(26)+Thm. 7.2-4

Thus it is proven by induction that

(31) For any prefix X of Ω, $|\Xi| \leq |X| \leq |\Delta|$, d is enabled in $S \cdot X$ and if

     d is a merge gate and b is it T (F) input arc, then d's control

input arc holds a <u>true</u> (<u>false</u>) token in $S \cdot X$,

(32) if there is a token on b in $S \cdot \Xi$, it is on b in $S \cdot X$, and

(33) there are the same number of firings of d in Δ as in Ξ, which is k-1

(34) Since $g \epsilon \alpha h$, $|\delta| \leq |\alpha|$, so $|\Phi(\delta)| \leq |\Phi(\alpha)|$ $\qquad$ (3)+Def. 4.3-4

(35) $|\Xi| \leq |\theta| \leq |\Delta|$ $\qquad$ (3)+(34)+(7)+Def. 2.4-5

(36) A $\qquad$ (35)+(31)

(37) B $\qquad$ (35)+(33)

(38) C $\qquad$ (35)+(14)+(32)

(39) D $\qquad$ (35)+(13)

$\triangle$

<u>Corollary 7.2-1</u> Given any program P, let S be any initial modified state of P and let θ be any firing sequence starting in S. Let d_1, d_2, ..., d_m be any ordered collection of distinct actors in P, all of which are enabled in $S \cdot \theta$. Then

A: $\theta \varphi_1 \varphi_2 \cdots \varphi_m$, where for i=1,...,m, φ_i is a firing of d_i, is a firing sequence starting in S, and

B: for i = 1,...,m, each token on an input arc of d_i in $S \cdot \theta$ is on that arc in $S \cdot \theta \varphi_1 \cdots \varphi_{i-1}$.

<u>Proof</u>: By induction on the index of the actors in the collection d_1, d_2, ..., d_m. Induction hypotheses are: For any n, $1 \leq n \leq m$,

A: $\theta \varphi_1 \cdots \varphi_n$ is a firing sequence starting in S,

B: for i=1,...,n, for j=i,...,m, each token on an input arc of d_j in $S \cdot \theta$ is on that arc in $S \cdot \theta \varphi_1, \ldots, \varphi_{i-1}$, and

C: For i=n,...,m, d_i is enabled in $S \cdot \theta \varphi_1 \cdots \varphi_{n-1}$.

Basis: n = 1.

(1) d_i is enabled in $S \cdot \theta$ for i=n,...,m. $\theta \varphi_1$, where φ_1 is a firing of

d_1, is a firing sequence starting in S. For i=1, for j=1,...,m,

each token on an input arc of d_j in $S \cdot \theta$ is on that arc in $S \cdot \theta$

Def. 2.3-1

Induction step: Assume that the induction hypotheses are true for any n,

$1 \leq n < m$.

(2) For i=n,...,m, d_i is enabled in $S \cdot \theta\varphi_1...\varphi_{n-1}$, and $\theta\varphi_1...\varphi_n$ is a

firing sequence starting in S ind. hyp.

(3) For i=n+1,...,m, d_i is enabled in $S \cdot \theta\varphi_1...\varphi_n$ (2)+Thm. 7.2-4

(4) $\theta\varphi_1...\varphi_n\varphi_{n+1}$ is a firing sequence starting in S (3)+Def. 2.3-1

(5) For i=1,...,n, for j=1,...,m, each token on an input arc of d_j in

$S \cdot \theta$ is on that arc in $S \cdot \theta\varphi_1...\varphi_{i-1}$ ind. hyp.

(6) For i=n+1, for j=i,...,m, each token on an input arc of d_j in $S \cdot \theta$

is on that arc in $S \cdot \theta\varphi_1...\varphi_{i-2}$ and then in $S \cdot \theta\varphi_1...\varphi_{i-2}\varphi_{i-1}$

(5)+(2)+Thm. 7.2-4

(7) For i=1,...,n+1, for j=i,...,m, each token on an input arc of d_j

in $S \cdot \theta$ is on that arc in $S \cdot \theta\varphi_1...\varphi_{i-1}$ (5)+(6)

Thus it is proven by induction that A and B are true for n=m; i.e.,

$\theta\varphi_1...\varphi_m$ is a firing sequence starting in S, and for i=1,...,m, each

token on an input arc of d_i (in particular) in $S \cdot \theta$ is on that arc in

$S \cdot \theta\varphi_1...\varphi_{i-1}$ $\triangle$

Lemma 7.2-8 For any program P, let S be any initial state for P, and let

Ω be any halted firing sequence starting in S. Let ω be any causal per-

mutation of $\eta(S,\Omega)$ such that $\Phi(\omega)$ is the reduction of Ω. If

(1) For every prefix εk of ω, letting Ex(d,m) be the target of k and

letting Δ be the prefix of Ω whose reduction is $\Phi(\varepsilon)$, there is an

initial state S' for P and a halted firing sequence Ω' starting in

S', and there is a prefix $\varepsilon'k'$ of some $\beta \in J_{S',\Omega'}$, such that $T(k') = T(k)$

and, for Δ' the prefix of Ω' whose reduction is $\Phi(\varepsilon')$, $S' \cdot \Delta'$ equals

either $S \cdot \Delta$ or $S \cdot \Xi$ where $\Delta = \Xi\varphi$ and φ is not a firing of d,

then ω is in $J_{S,\Omega}$.

Proof:

(2) Let εk be any prefix of ω, let the destination in $T(k)$ be

Dst(Ex(d,m),j), and let Δ be the prefix of Ω whose reduction is

$\Phi(\varepsilon)$. Then there is an initial state S' for P and a halted firing

sequence Ω' starting in S', and a prefix $\varepsilon'k'$ of some $\beta \in J_{S',\Omega'}$

such that $T(k') = T(k)$ and, for Δ' the prefix of Ω' whose reduc-

tion is $\Phi(\varepsilon')$, $S' \cdot \Delta'$ equals either $S \cdot \Delta$ or $S \cdot \Xi$ where $\Delta = \Xi\varphi$ and φ

is not a firing of d   (1)

(3) If $\Delta = \Xi\varphi$ and φ is not a firing of d, then every input arc of d

which holds a token in $S \cdot \Xi$ holds the same token in $S \cdot \Delta$, and d is

enabled in $S \cdot \Xi \Rightarrow$ d is enabled in $S \cdot \Delta$ (2)+Thm. 7.2-4

(4) Every actor is enabled in $S' \cdot \Delta'$ iff it is enabled in any state

equal to $S' \cdot \Delta'$ Cor. 7.1-2

(5) Every control arc holds only non-pointer-valued tokens   Def. 2.2-1

(6) Every control arc holds tokens of the same value in two equal

states   (5)+Defs. 7.1-2+3.4-1

(7) If $S \cdot \Delta$ equals $S' \cdot \Delta'$, then d is a gate $\Rightarrow$ its control input arc holds

the same token in $S \cdot \Delta$ as in $S' \cdot \Delta'$, and d is enabled in $S' \cdot \Delta' \Rightarrow$

d is enabled in $S \cdot \Delta$. If $S \cdot \Xi$ is equal to $S' \cdot \Delta'$ and φ is not a

firing of d, then d is enabled in $S' \cdot \Delta' \Rightarrow$ d is enabled in $S \cdot \Xi \Rightarrow$

d is enabled in $S \cdot \Delta$ $\wedge$ if d is a gate, its control input arc holds

a token in $S' \cdot \Delta'$ $\Rightarrow$ that arc holds the same token in $S \cdot \Xi$, hence in

$S \cdot \Delta$ (6)+(4)+(3)+Defs. 3.3-6+2.1-4

(8) If d$\notin$DL, let b be the number-j input arc of the actor labelled d.

That actor is enabled in $S' \cdot \Delta'$, hence in $S \cdot \Delta$, and if it is a

merge gate and b is its T (F) input arc, then its control input

arc holds a <u>true</u> (<u>false</u>) token in $S' \cdot \Delta'$, hence in $S \cdot \Delta$

                                     (2)+(7)+Def. 4.3-5

(9) Since N and V_p are infinite, for any firing sequence X, there is

an equal firing sequence X' such that the multiset of pointer-node

pairs in the Copy firings in X' is consistent with the heap

in $S' \cdot \Delta'$ Defs. 2.2-1+2.4-5+5.2-3

(10) For any firing sequence starting in any state equal to $S' \cdot \Delta'$, there

is an equal firing sequence starting in $S' \cdot \Delta'$ (9)+Cor. 7.1-2

(11) If $S \cdot \Delta$ equals $S' \cdot \Delta'$, there is a firing sequence starting in $S \cdot \Delta$ which

contains a firing of actor c $\Rightarrow$ there is a firing sequence starting

in $S' \cdot \Delta'$ which contains a firing of c. If $S \cdot \Xi$ equals $S' \cdot \Delta'$, there

is a firing sequence X starting in $S \cdot \Delta$ which contains a firing of

c $\Rightarrow$ there is a firing sequence φX starting in $S \cdot \Xi$ which contains

a firing of c $\Rightarrow$ there is a firing sequence starting in $S' \cdot \Delta'$

which contains a firing of c   (10)+Defs. 2.4-5+2.3-1

(12) d$\in$DL and d = (c,n) $\Rightarrow$ letting b be the number-n program output arc

of P if c = "OD", or else the number-n input arc of the actor

labelled c, there is a token on b in $S' \cdot \Delta'$, and [c is an actor

label $\Rightarrow$ there is no firing sequence starting in $S' \cdot \Delta'$ which

contains a firing of c]   (2)+Def. 4.3-5

(13) ⟹ there is a token on b in $S \cdot \Delta$, if $S \cdot \Delta$ equals $S' \cdot \Delta'$, or in $S \cdot \Xi$,

if $S \cdot \Xi$ equals $S' \cdot \Delta'$ (2)+Defs. 7.1-2+3.4-1

(14) ∧ [c is an actor label ⟹ there is no firing sequence starting in

$S \cdot \Delta$ which contains a firing of c, and if $S \cdot \Xi$ equals $S' \cdot \Delta'$, φ is

not a firing of c]   (2)+(11)

(15) ⟹ there is a token on b in $S \cdot \Delta$ Thm. 7.2-4

(16) ω is in $J_{S,\Omega}$ (2)+(8)+(12)+(14)+(15)+Def. 4.3-5

$\triangle$

Theorem 7.2-5 Every expansion (Int,J) from $EE(L_D,M)$ satisfies the Persistence Axiom.

Proof:

(1) Let J be any job in J. There is an L_D program P such that Int is

Int(P), and there is an equivalence class E of initial modified

states of P such that $J = J_E$ Defs. 4.3-1+4.3-2

(2) Let αg be any computation in J. There is an initial modified state

$S \in E$ and a halted firing sequence Ω starting in S such that αg is

a prefix of some β in $J_{S,\Omega}$ (1)+Def. 4.3-3

(3) Let θ be the prefix of Ω whose length equals that of $\Phi(\alpha)$. Then

θ is a firing sequence starting in S whose reduction is $\Phi(\alpha)$

                                      Lemma 7.2-2+Def. 2.3-1

(4) Let Int(P) = (St, I,IE). Let EF be the set of executions {Ex(d,k)|

d∈St-DL and Ex(d,k) is not initiated in α but has an input entry

in αg}. Let e = Ex(d,k) be any execution in EF. Let Δφ be the

prefix of Ω in which φ is the kth firing of d. Let f = Ent(e,j)

be any input entry to e in αg and let b be d's number-j input

arc. Then

(4a) d is enabled in $S \cdot \theta$, and if d is a merge gate and b is its T (F)

   input arc, then d's control input arc holds a <u>true</u> (<u>false</u>) token

   in $S \cdot \theta$,

(4b) there are exactly k-1 firings of d in θ,

(4c) there is a token on b in $S \cdot \Delta$ which is on b in $S \cdot \theta$, and

(4d) b is an output arc of an actor c $\Rightarrow$ there are the same number of

   firings of c in θ as in Δ (2)+(3)+Lemma 7.2-7

 (5) Let t be any transfer in $ET_J(\alpha)$ except $T(g)$. Then there is an

   entry h with $T(h) = t$ such that αh is in J   Def. 6.2-2

 (6) There is an initial modified state $S' \in E$ and a halted firing sequence

   Ω' starting in S' such that αh is a prefix of some $\beta' \in J_{S', \Omega'}$

                                     (1)+(5)+Def. 4.3-3

 (7) Let θ' be the prefix of Ω' whose length equals that of $\Phi(\alpha)$. Then

   θ' is a firing sequence starting in S' whose reduction is $\Phi(\alpha)$

                               (6)+Lemma 7.2-2+Def. 2.3-1

 (8) Let $Dst(e', j')$ where $e' = Ex(d', k')$ be the destination in $T(h)$.

   Then e' has an input entry in αh but is not initiated in α

                                     Defs. 4.2-5+4.2-6

 (9) If $d' \in St-DL$, let $\Delta' \varphi'$ be the prefix of Ω' in which φ' is the k'^{th}

   firing of d', and let b' be the number-j' input arc of d'. Then

(9a) d' is enabled in $S' \cdot \theta'$, and if d' is a merge gate and b' is its

   T (F) input arc, its control input arc holds a <u>true</u> (<u>false</u>) token

   in $S' \cdot \theta'$,

(9b) there are exactly k'-1 firings of d' in θ',

(9c) there is a token on b' in $S' \cdot \Delta'$ which is on b' in $S' \cdot \theta'$, and

(9d) b' is an output arc of an actor c' $\Rightarrow$ there are the same number of

firings of c' in θ' as in Δ'   (4)+(6)+(7)+(8)+Lemma 7.2-7

(10) S' equals S (1)+(2)+(6)

(11) θ' equals θ   (3)+(7)+Def. 2.4-5

(12) $S' \cdot \theta'$ equals $S \cdot \theta$ (10)+(11)+Thm. 7.1-2

(13) d' is enabled in $S \cdot \theta$ (12)+(9a)+Cor. 7.1-2

(14) If d' is a merge gate and b' is its T (F) input arc, then its

control input arc holds a <u>true</u> (<u>false</u>) token in $S \cdot \theta$

(9a)+(12)+Defs. 7.1-2+3.4-1

(15) There are exactly k'-1 firings of d' in θ   (9b)+(11)+Def. 2.4-5

(16) Let EA be the set $\{d| \exists k: Ex(d,k) \in EF\}$, if d'∈DL, or $\{d| \exists k:$

$Ex(d,k) \in EF\} \cup \{d'\}$, if d'∈St-DL. Then each actor in EA is enabled

in $S \cdot \theta$ (4)+(13)

(17) Let d_1, d_2, ..., d_m be any ordering of the actors in EA satisfying

the following: If g initiates Ex(d",k"), then d_1 = d" and d_2 = d',

otherwise, d_1 = d'

(18) $\theta \varphi_1 \varphi_2 \cdots \varphi_m$, where for i=1,...,m, φ_i is a firing of d_i, is a firing

sequence starting in S, and for i=1,...,m, each token on an input

arc of d_i in $S \cdot \theta$ is on that arc in $S \cdot \theta \varphi_1 \cdots \varphi_{i-1}$

(2)+(3)+(16)+(17)+Cor. 7.2-1

(19) β is a causal permutation of η(S,Ω) and β' is a causal permutation

of η(S',Ω')   (2)+(6)+Def. 4.3-5

(20) Let χ be any halted firing sequence starting in S which has

$\theta \varphi_1 \cdots \varphi_m$ as a prefix. Let AF be the set of executions $\{Ex(d,k)|$

d∈St-DL and Ex(d,k) is initiated in α}. Let f be any entry in αg.

Then f is in η(S,Ω)   (2)+(19)

(21) Let e = Ex(d,k) be the target of f. Then d∈St-DL ⟹ e∈AF∪EF (4)+(20)

(22) $\wedge$ letting $V(f)$ be v and $T(f)$ be $(s,Dst(Ex(d,k),j))$, and letting

    b be the number-j input arc of d, there is a prefix $\Delta\varphi$ of Ω in

    which φ is the kth firing of d, φ removes a token of value v

    from b, $s = Source(b,S,\Delta)$, and f is in $\eta(S,\Delta\varphi)$ (20)+Alg. 4.3-1

(23) $d\in St-DL\wedge e\in AF \Rightarrow$ there are at least k firings of d in θ

                                                                         (1)+(2)+(19)+(3)+Thm. 4.3-2

(24) $\Rightarrow \Delta\varphi$ is a prefix of θ, hence of X (22)+(20)

(25) $\Rightarrow$ there is a prefix $\Delta\varphi$ of X such that f is in $\eta(S,\Delta\varphi)$ (22)

(26) $\Rightarrow f$ is in $\eta(S,X)$ Alg. 4.3-1

(27) $d\in St-DL\wedge e\in EF \Rightarrow d\in EA$ (16)

(28) $\Rightarrow$ there is an i such that $d = d_i$ in the ordering of the actors in

    EA   (17)

(29) $\Rightarrow$ in $\theta\varphi_1\ldots\varphi_m$, φ_1 is a firing of d (18)

(30) $d\in St-DL\wedge e\in EF \Rightarrow$ since all actors in EA are distinct, φ_i is the kth

    firing of d in X (27)+(29)+(4b)+(20)

(31) $\wedge$ [d is a merge gate and b is its T (F) input arc $\Rightarrow d$'s control

    input arc holds a <u>true</u> (<u>false</u>) token in $S\cdot\theta$ (4a)

(32) $\Rightarrow d$'s control input arc holds a <u>true</u> (<u>false</u>) token in $S\cdot\theta\varphi_1\ldots\varphi_{i-1}$]

                                                    (28)+(18)

(33) $\Rightarrow$ there is a token on b in $S\cdot\theta$ (27)+(16)+(31)+Defs. 3.3-6+2.1-4

(34) $\Rightarrow$ there is a token on b in $S\cdot\theta\varphi_1\ldots\varphi_{i-1}$ identical to the one on b

    in $S\cdot\Delta$, and it is removed by φ_i

                            (28)+(4c)+(18)+(31)+(32)+Defs. 3.3-9+2.1-5

(35) $\wedge$ [b is an output arc of an actor $c \Rightarrow c$ is not enabled in $S\cdot\theta$

                                              Defs. 3.3-6+2.1-4

(36) $\Rightarrow c\notin EA$ (16)

(37) $\Rightarrow$ the number of firings of c in $\theta\varphi_1\ldots\varphi_{i-1}$ is equal to the number

of firings of c in θ (18)+(17)

(38) which is equal to the number of firings of c in Δ]   (30)+(4d)

(39) S equals S Defs. 7.1-2+3.4-1

(40) $d \in St-DL \wedge e \in EF \Rightarrow Source(b,S,\theta\varphi_1\ldots\varphi_{i-1}) = Source(b,S,\Delta) = s$

                                (39)+(3)+(18)+(35)+(37)+(22)+(38)+Lemma 7.1-3

(41) $\Rightarrow$ f is in $\eta(S,\chi)$ (30)+(34)+(22)+Alg. 4.3-1

(42) For any entry $f \in ag$ whose target is $Ex(d,k)$, $d \in St-DL \Rightarrow$ f is in $\eta(S,\chi)$

                                     (20)+(21)+(23)+(26)+(40)+(41)

(43) h is in $\eta(S',\Omega')$ (6)+(19)

(44) Let $Ex(d',k')$ be the target of h. Then $d' \in St-DL \Rightarrow d' \in EA$ (16)

(45) $\wedge$ letting $T(h)$ be $(s,Dst(Ex(d',k'),j'))$, there is a prefix $\Delta'\varphi'$ of

Ω' in which φ' is the k'^{th} firing of d', φ' removes a token from

b', the number-j' input arc of d', and $s = Source(b',S',\Delta')$

                                          (43)+Alg. 4.3-1

(46) $\Rightarrow$ there is an i such that $d = d_i$ in the ordering of actors in EA(17)

(47) $\Rightarrow$ in $\theta\varphi_1\ldots\varphi_m$, φ_i is a firing of d'   (18)

(48) $\Rightarrow$ since each actor in EA is distinct, φ_i is the k'^{th} firing of d'

in χ (45)+(8)+(15)+(20)

(49) $\wedge$ [d' is a merge gate and b' is its T (F) input arc $\Rightarrow$ its control

input arc holds a <u>true</u> (<u>false</u>) token in $S \cdot \theta$ (45)+(18)+(14)

(50) $\Rightarrow$ its control input arc holds a <u>true</u> (<u>false</u>) token in $S \cdot \theta\varphi_1\ldots\varphi_{i-1}$]

                                            (46)+(18)

(51) $\Rightarrow$ there is a token on b' in $S \cdot \theta$ (44)+(16)+(49)+Defs. 3.3-6+2.1-4

(52) $\Rightarrow$ there is a token on b' in $S \cdot \theta\varphi_1\ldots\varphi_{i-1}$, and it is removed by φ_i

                                   (46)+(18)+(49)+(50)+Defs. 3.3-9+2.1-5

(53) $\wedge$ [b' is an output arc of an actor c $\Rightarrow$ ċ is not enabled in $S \cdot \theta$

Defs. 3.3-6+2.1-4

(54) $\Rightarrow$ c$\notin$EA $\Rightarrow$ the number of firings of c in $\theta \varphi_1 \cdots \varphi_{i-1}$ equals the number

of firings of c in θ (16)+(18)+(17)

(55) which equals the number of firings of c in θ' (11)+Def. 2.4-5

(56) which equals the number of firings of c in Δ'] (45)+(8)+(9d)

(57) $\Rightarrow$ Source(b',S,$\theta \varphi_1 \cdots \varphi_{i-1}$) = Source(b',$S'$,$\Delta'$) = s

(10)+(7)+(18)+(53)+(54)+(56)+(45)+Lemma 7.1-3

(58) $\Rightarrow$ there is an entry in $\eta(S,X)$ with transfer (s,Dst(Ex(d',k'),j')),

which is T(h) (48)+(52)+(45)+Alg. 4.3-1

(59) Let f be any entry in αg, and let δ be such that δf is a prefix of

αg, hence β. Let Ξ be the prefix of Ω whose length equals that

of $\Phi(\delta)$. Then Ξ is a firing sequence starting in S whose reduc-

tion is $\Phi(\delta)$ (2)+Lemma 7.2-2+Def. 2.3-1

(60) $|\Phi(\delta)| \leq |\Phi(\alpha)|$ (59)+Def. 4.3-4

(61) Ξ is a prefix of θ, hence of X (3)+(59)+(60)+(20)

(62) Let Ex(d,k) be the target of f. d$\notin$St-DL $\Rightarrow$ d = (c',n) where c' is

either the label of an actor in P or "OD", there is an arc b

uniquely related to d which holds a token of value V(f) in $S \cdot \Omega$,

and T(f) = (s,Dst(Ex(d,0),1)) where s = Source(b,S,Ω)(20)+Alg. 4.3-1

(63) $\wedge$ there is a token on b in $S \cdot \Xi$ $\wedge$ [c' is the label of an actor in P

$\Rightarrow$ in no firing sequence $\Delta \varphi$ starting in $S \cdot \Xi$ is φ a firing of c']

(59)+(2)+Def. 4.3-5

(64) $\Rightarrow$ [c' is the label of an actor in P $\Rightarrow$ b is an input arc of c'

Alg. 4.3-1

(65) $\Rightarrow$ for no firing sequence $\Delta \varphi$ starting in $S \cdot \Xi$ is there a token on b

in $S \cdot \Xi \Delta$ but none in $S \cdot \Xi \Delta \varphi$]   (63)+Def. 3.3-9+2.1-5

(66) $\wedge$ [c' is not the label of an actor in P $\Rightarrow$ b is a program output arc

Alg. 4.3-1

(67) $\Rightarrow$ b is not an input arc of any actor   Def. 2.1-1

(68) $\Rightarrow$ for no firing sequence $\Delta \varphi$ starting in $S \cdot \Xi$ is there is a token on

b in $S \cdot \Xi \Delta$ but none in $S \cdot \Xi \Delta \varphi$]   Defs. 3.3-9+2.1-5

(69) $\Rightarrow$ for any firing sequence $\Delta \varphi$ starting in $S \cdot \Xi$, there is a token on

b in $S \cdot \Xi \Delta \varphi$ (63)

(70) $\Rightarrow$ there is a token on b in $S \cdot \chi$ (61)

(71) $\wedge$ [b is an output arc of actor c $\Rightarrow$ for every firing sequence Δ

starting in $S \cdot \Xi$, c is not enabled in $S \cdot \Xi \Delta$ Defs. 3.3-6+2.1-4

(72) $\wedge$ there is no p such that $c \in Q(p)$ in $S \cdot \Xi \Delta$ Cor. 7.1-1

(73) $\Rightarrow$ there are the same number of firings of c in Ξ as in Ω, and there

are the same number of firings of c in Ξ as in χ]   (59)+(61)

(74) $\Rightarrow$ Source(b,S,χ) = Source(b,S,Ω) = s   (39)+(62)+Lemma 7.1-3

(75) $\wedge$ for every firing sequence $\Delta \varphi$ starting in $S \cdot \Xi$, any token on b in

$S \cdot \Xi \Delta \varphi$ is on b in $S \cdot \Xi \Delta$ (71)+(72)+Defs. 3.3-9+2.1-5

(76) $\Rightarrow$ there is a token of value v on b in $S \cdot \Xi$, hence $S \cdot \chi$ (62)+(59)+(61)

(77) $\Rightarrow$ f is in $\eta(S,\chi)$ (62)+(70)+(74)+Alg. 4.3-1

(78) For any actor c, there is a firing sequence $\Delta \varphi$ starting in $S \cdot \theta$ in

which φ is a firing of c $\Rightarrow$ since N and V_p are infinite, there is

an equal firing sequence $\Delta' \varphi'$ such that the multiset of pointer-

node pairs in the Copy firings in $\Delta' \varphi'$ is consistent with the

heap in $S' \cdot \theta'$ Defs. 2.2-1+2.4-5+5.2-3

(79) $\Rightarrow$ there is a firing sequence $\Delta' \varphi'$ starting in $S' \cdot \theta'$ in which φ' is

a firing of c   (12)+Cor. 7.1-2+Def. 2.4-5

(80) Let Ex(d,k) be the target of h. d∉St-DL ⇒ d = (c',n) where c' is

  either the label of an actor in P or "OD", there is an arc b

  uniquely related to d which holds a token in $S' \cdot \Omega'$, and T(h) is

  (s,Dst(Ex(d,0),1)) where s = Source(b,S',Ω')   (43)+Alg. 4.3-1

(81) ∧ there is a token on b in $S' \cdot \theta'$ ∧ [c' is the label of an actor in

  P ⇒ in no firing sequence Δφ starting in $S' \cdot \theta'$ is φ a firing of c]

                                    (6)+(7)+Def. 4.3-5

(82) ⇒ there is a token on b in $S \cdot \theta$ (12)+Defs. 7.1-2+3.4-1

(83) ∧ [c' is the label of an actor ⇒ there is no firing sequence Δφ

  starting in $S \cdot \theta$ in which φ is a firing of c']   (78)+(79)

(84) ⇒ [c' is the label of an actor ⇒ b is an input arc of c'

                                    (80)+Alg. 4.3-1

(85) ⇒ for no firing sequence Δφ starting in $S' \cdot \theta'$ ($S \cdot \theta$) is there a

  token on b in $S' \cdot \theta' \Delta$ ($S \cdot \theta \Delta$) but none in $S' \cdot \theta' \Delta\varphi$ ($S \cdot \theta \Delta\varphi$)]

                                    (81)+(83)+Defs. 3.3-9+2.1-5

(86) ∧ [c' is not the label of an actor ⇒ b is a program output arc

                                    (80)+Alg. 4.3-1

(87) ⇒ b is not an input arc of any actor   Def. 2.1-1

(88) ⇒ for no firing sequence Δφ starting in $S' \cdot \theta'$ ($S \cdot \theta$) is there a

  token on b in $S' \cdot \theta' \Delta$ ($S \cdot \theta \Delta$) but none in $S' \cdot \theta' \Delta\varphi$ ($S \cdot \theta \Delta\varphi$)]

                                    Defs. 3.3-9+2.1-5

(89) ⇒ for every firing sequence Δφ starting in $S' \cdot \theta'$ ($S \cdot \theta$), there is

  a token on b in $S' \cdot \theta' \Delta\varphi$ ($S \cdot \theta \Delta\varphi$)   (81)+(82)

(90) ⇒ there is a token on b in $S \cdot \chi$ (20)

(91) ∧ [b is an output arc of actor c ⇒ for no firing sequence Δ starting

  in $S' \cdot \theta'$ ($S \cdot \theta$) is c enabled in $S' \cdot \theta' \Delta$ ($S \cdot \theta \Delta$)   Defs. 3.3-6+2.1-4

(92) $\Rightarrow$ there are the same number of firings of c in θ' as in Ω' and the

same number of firings of c in θ as in χ (7)+(20)

(93) $\Rightarrow$ there are the same number of firings of c in χ as in Ω']

                                    (11)+Def. 2.4-5

(94) $\Rightarrow$ Source(b,S,χ) = Source(b,S',Ω') = s

                         (10)+(20)+(6)+(80)+(90)+Lemma 7.1-3

(95) $\Rightarrow$ there is an entry in $\eta(S,\chi)$ with transfer (s,Dst(Ex(d,0),1)),

which is T(h)   (20)+(90)+(89)+Alg. 4.3-1

(96) Let $\bar{h}$ be the entry in $\eta(S,\chi)$ such that $T(\bar{h}) = T(h)$. Let δ be the

sequence of entries derived by striking every entry in αg, plus $\bar{h}$,

from $\eta(S,\chi)$. Then $\alpha g \bar{h} \delta$ is a permutation of $\eta(S,\chi)$, and for any

two entries f_1 and f_2 in δ, f_1 follows f_2 in δ iff f_1 follows f_2

in $\eta(S,\chi)$ (42)+(62)+(77)+(44)+(58)+(80)+(95)+Def. 4.2-6

(97) Let m be the length of $\Phi(\alpha g \bar{h})$. $\Phi(\alpha g \bar{h})$ is $\Phi(\alpha)$ followed by zero,

one, or two firings   Def. 4.3-4

(98) $\Phi(\alpha g \bar{h}) = \Phi(\alpha) \Rightarrow$ the reduction of θ is $\Phi(\alpha g \bar{h})$ and θ is a prefix

of χ (3)+(20)

(99) $\Phi(\alpha g \bar{h}) = \Phi(\alpha)\varphi_a \Rightarrow$ one of g and $\bar{h}$ is an initiating entry and [g

initiates an execution Ex(d'',k'') $\Rightarrow \varphi_a$ is a firing of d'' Def. 4.3-4

(100) $\wedge \varphi_1$ is a firing of d'']   (17)+(18)

(101) $\wedge$ [$\bar{h}$ initiates an execution Ex(d',k') in $\alpha g \bar{h} \Rightarrow$ g does not initiate

an execution in αg and φ_a is a firing of d'   Defs. 4.2-6+4.3-4

(102) $\Rightarrow \varphi_1$ is a firing of d']   (17)+(18)

(103) $\wedge$ the reduction of $\theta \varphi_1$ is one firing longer than the reduction of

θ, which is $\Phi(\alpha)$, so it is $\Phi(\alpha)\varphi_a$ where ψ_a is a firing of the

same actor as φ_1 (3)+(96)+Defs. 2.4-5+4.3-4

(104) $\Rightarrow$ the reduction of $\theta\varphi_1$ is $\Phi(\alpha g\bar{h})$ and $\theta\varphi_1$ is a prefix of χ (99)+(20)

(105) $\Phi(\alpha g\bar{h}) = \Phi(\alpha)\varphi_a\varphi_b \Rightarrow$ both of g and $\bar{h}$ are initiating entries

Def. 4.3-4

(106) $\Rightarrow$ g initiates an execution Ex(d",k") in $\alpha g\bar{h}$ and g initiates an

execution Ex(d",k") in αg (96)+Def. 4.3-4

(107) $\Rightarrow \varphi_a$ is a firing of d", as is φ_1, and φ_b is a firing of d', as is

φ_2 (17)+(18)+Def. 4.3-4

(108) $\wedge$ the reduction of $\theta\varphi_1\varphi_2$ is two firings longer than the reduction

of θ, which is $\Phi(\alpha)$, so it is $\Phi(\alpha)\varphi_a\varphi_b$ where φ_a is a firing of the

same actor as φ_1 and φ_b is a firing of the same actor as φ_2

(3)+(96)+Defs. 2.4-5+4.3-4

(109) $\Rightarrow$ the reduction of $\theta\varphi_1\varphi_2$ is $\Phi(\alpha g\bar{h})$ and $\theta\varphi_1\varphi_2$ is a prefix of χ

(105)+(20)

(110) $\Phi(\alpha g\bar{h})$ is the reduction of the prefix Λ of χ whose length equals

that of $\Phi(\alpha g\bar{h})$, i.e., m (97)+(98)+(99)+(104)+(105)+(109)

(111) Let γf be any prefix of $\alpha g\bar{h}\delta$ and let e be the execution of which

f is an output entry. $f\epsilon\alpha g \Rightarrow$ there is a prefix γf of β in which

f is an output entry of e (2)

(112) $\Rightarrow$ e is initiated in γ (19)+Def. 4.2-7

(113) $f = \bar{h} \Rightarrow \alpha$ is a prefix of γ and h is an output entry of e $\Rightarrow$ there

are at least as many input entries to e in γ as in α(96)+Def. 4.2-5

(114) $\wedge$ e is initiated in α (6)+(19)+Def. 4.2-7

(115) $\Rightarrow$ e is initiated in γ Def. 4.2-6

(116) $\eta(S,\chi)$ is causal wrt Int and is a computation for Int

(1)+Lemma 4.3-1

(117) f is in $\eta(S,\chi)$ (111)+(96)

(118) Let e = Ex(d,k). In(l(d)) input entries to e precede f in $\eta\langle S, X\rangle$

(111)+(116)+(117)+(4)+Defs. 4.2-7+4.2-6

(119) There are In(l(d)) input entries to e in $\alpha g\bar{h}\delta$ (118)+(96)

(120) $f \in \delta \Rightarrow$ no entry follows f in $\alpha g\bar{h}\delta$ unless it follows f in $\eta(S, X)$ (96)

(121) $\Rightarrow$ In(l(d)) input entries to e precede f in $\alpha g\bar{h}\delta$ (119)+(118)

(122) $\Rightarrow$ e is initiated in γ (111)+Def. 4.2-6

(123) e is initiated in γ (111)+(112)+(113)+(115)+(120)+(122)

(124) $\alpha g\bar{h}\delta$ is a causal permutation of $\eta(S, X)$ (111)+(123)+(96)+Def. 4.2-7

(125) Let Z be the set of executions $\{Ex(d,k) \mid d \in St\text{-}DL$ and $Ex(d,k)$ is initiated in $\eta(S, X)\}$. Let Y be the set $\{Ex(d,k) \mid d \in St\text{-}DL$ and $Ex(d,k)$ is initiated in $\eta(S, \Lambda)\}$. Since $\eta(S, \Lambda)$ is a prefix of $\eta(S, X)$, every initiating entry to an execution in Z-Y is preceded in $\eta(S, X)$ by the initiating entries to all executions in Y

(110)+Alg. 4.3-1

(126) For any $d \in St\text{-}DL$, In(l(d)) > 0 Defs. 4.3-2+4.3-1+2.1-2

(127) For any e = Ex(d,k), $e \in Y$ iff there are In(l(d)) > 0 input entries to e in $\eta(S, \Lambda)$ and $d \in St\text{-}DL$ (125)+(126)+Def. 4.2-6

(128) iff there are at least k firings of d in Λ and $d \in St\text{-}DL$ Lemma 4.3-1

(129) iff e is initiated in $\alpha g\bar{h}$ (1)+(4)+(20)+(124)+(110)+Thm. 4.3-2

(130) iff at least In(l(d)) input entries to e are deleted from $\eta(S, X)$ to get δ (96)

(131) Z is also the set $\{Ex(d,k) \mid d \in St\text{-}DL$ and $Ex(d,k)$ is initiated in $\alpha g\bar{h}\delta\}$ (125)+(124)+Def. 4.2-6

(132) The number of initiating entries in $\alpha g\bar{h}$ to executions in Z equals the length of $\Phi(\alpha g\bar{h})$ which is m   (131)+(110)+Def. 4.3-4

(133) For n > m, let e be the nth execution from Z to initiate in $\alpha g\bar{h}\delta$.

Then f, the initiating entry of e in $\alpha g \bar{h} \delta$, is in δ (132)

(134) e is in Z-Y   (133)+(129)+(127)

(135) Let e' = Ex(d',k') be any other execution in Z and let f' be its

     initiating entry in $\alpha g \bar{h} \delta$. Then f' precedes f in $\alpha g \bar{h} \delta$ iff f' is

     in $\alpha g \bar{h}$, or f' precedes f in $\eta(S,X)$ and fewer than $In(I(d'))$ input

     entries to e' are deleted from $\eta(S,X)$ to get δ (96)+Def. 4.2-6

(136) iff e' is in Y, or f' precedes f in $\eta(S,X)$ and e'$\not\in$Y

                                               (127)+(129)+(130)

(137) iff e' is in Y and the initiating entry to e' precedes f in $\eta(S,X)$

     or e'$\not\in$Y and f' precedes f in $\eta(S,X)$ iff the initiating entry to

     e' precedes f in $\eta(S,X)$ (134)+(125)

(138) For n > m, the nth execution from Z initiated in $\alpha g \bar{h} \delta$ is the nth

     execution from Z initiated in $\eta(S,X)$ (135)+(137)

(139) $\Phi(\eta(S,X))$ is the reduction of X and $\eta(S,X)$ is in $J_{S,X}$

                                                  (20)+Lemma 4.3-3

(140) For any n > m, the nth firing in the reduction of X is a firing

     of d iff the nth firing in $\Phi(\eta(S,X))$ is a firing of d   (139)

(141) iff the nth execution from Z initiated in $\eta(S,X)$ is an execution

     of d   (125)+Def. 4.3-4

(142) iff the nth execution from Z initiated in $\alpha g \bar{h} \delta$ is an execution of

     d   (138)

(143) iff the nth firing in $\Phi(\alpha g \bar{h} \delta)$ is a firing of d   (125)+Def. 4.3-4

(144) For any n $\leq$ m, the nth firing in the reduction of X is the nth

     firing in the reduction of Λ (110)+Def. 2.4-5

(145) which is the nth firing in $\Phi(\alpha g \bar{h})$ (110)

(146) which is the nth firing in $\Phi(\alpha g \bar{h} \delta)$ Def. 4.3-4

(147) $\Phi(\alpha g\bar{h}\delta)$ is the reduction of X (140)+(143)+(144)+(146)+Def. 2.4-5

(148) Let γf be any prefix of $\alpha g\bar{h}\delta$, let Δ be the prefix of X whose

reduction is $\Phi(\gamma)$, and let the destination in $T(f)$ be $Dst(e,j)$

where $e = Ex(d,m)$ (147)+Lemma 7.2-2

(149) $f \in \alpha g \Rightarrow$ there is a prefix γf of $\beta \in J_{S,\Omega}$ such that, for Δ' the prefix

of Ω whose reduction is $\Phi(\gamma)$, Δ equals Δ' (2)+Def. 2.4-5

(150) $\Rightarrow S \cdot \Delta$ equals $S \cdot \Delta'$ (39)+Thm. 7.1-2

(151) All $In(I(d))$ input entries to e are consecutive in $\eta(S,X)$

     (148)+Alg. 4.3-1

(152) All input entries to e which are left in δ are consecutive

     (151)+(96)

(153) $f \in \delta \Rightarrow$ there is a prefix $\gamma' f$ of $\eta(S,X) \in J_{S,X}$ (96)+(139)

(154) $\Rightarrow$ for any execution $e' \neq e$ from Z, the initiating entry f' to e'

precedes f in γf (i.e., is in γ) iff it precedes e's initiating

entry in $\alpha g\bar{h}\delta$ (148)+(152)

(155) iff the initiating entry to e' precedes the initiating entry to

e in $\eta(S,X)$ (133)+(135)+(137)

(156) iff the initiating entry to e' precedes f in $\eta(S,X)$ (i.e., is in

γ')   (151)+(153)

(157) $\Rightarrow \exists n: |\Phi(\gamma')| = |\Phi(\gamma)| = n$ (125)+Def. 4.3-4

(158) $\Rightarrow \Phi(\gamma')$ is the length-n prefix of $\Phi(\eta(S,X))$, which is the

reduction of X (139)+(153)+Def. 4.3-4

(159) $\wedge \Phi(\gamma)$ is the length-n prefix of $\Phi(\alpha g\bar{h}\delta)$, which is the reduction

of X (147)+(148)+Def. 4.3-4

(160) $\Rightarrow$ for Δ' the prefix of X whose reduction is $\Phi(\gamma')$, Δ' equals Δ

     (148)+Def. 2.4-5

(161) $\Rightarrow S \cdot \Delta'$ equals $S \cdot \Delta$ (39)+Thm. 7.1-2

(162) Ex(d,m) is not initiated in γ (148)+Def. 4.2-6

(163) $f = \bar{h} \Rightarrow \gamma = \alpha g \Rightarrow$ if g is the initiating entry in $\alpha g \bar{h}$ of an execution

d''$\in$St-DL, then d'' $\neq$ d, so $\Phi(\gamma) = \Phi(\alpha)\varphi$, where φ is not a firing

of d, otherwise $\Phi(\gamma) = \Phi(\alpha)$ (162)+Def. 4.3-4

(164) $\Rightarrow$ there is a prefix αh of $\beta' \in J_{S',\Omega'}$ such that $T(f) = T(\bar{h}) = T(h)$

and $\Phi(\gamma)$ equals either $\Phi(\alpha)$ or $\Phi(\alpha)\varphi$ where φ is not a firing of d

         (96)+Def. 4.3-4

(165) $\Rightarrow$ letting Δ' be the prefix of Ω' whose reduction is $\Phi(\alpha)$, Δ'

equals either Δ or θ where $\Delta = \theta\varphi$ and φ is not a firing of d

         (148)+Def. 2.4-5

(166) $\Rightarrow S' \cdot \Delta'$ equals either $S \cdot \Delta$ or $S \cdot \theta$ where $\Delta = \theta\varphi$ and φ is not a

firing of d   (10)+Thm. 7.1-2

(167) $\alpha g \bar{h}\delta$ is in $J_{S,\chi}$ (124)+(147)+(148)+(149)+(2)+(150)+(153)+(20)+(160)+

         (161)+(163)+(164)+(6)+(165)+(166)+Lemma 7.2-8

(168) $\alpha g \bar{h}$ is in J   (167)+(2)+(20)+(1)+Def. 4.3-3

(169) $T(\bar{h}) = T(h) = t$ is in $ET_J(\alpha g)$ (168)+(96)+Def. 6.2-2

(170) (Int,J) satisfies the Persistence Axiom (1)+(2)+(5)+(169)+Ax. 6.2-5

$\triangle$

7.3 Determinacy and Functionality

The preceding two sections have proven that $EE(L_D,M)$ is an S-S model

and that every expansion from it satisfies the Determinacy Axioms. By

Theorem 6.4-1, then, every expansion is determinate. This section

contains the final and most complex proof of the thesis, that if the

expansion (Int,J) of an L_D program P is determinate, then P running on

the modified interpreter is functional.

P is functional iff for any two equal initial modified states S_1 and S_2 for P, and any two halted firing sequences Ω_1 and Ω_2 starting in S_1 and S_2, $S_2 \cdot \Omega_2$ equals $S_1 \cdot \Omega_1$. There is an equivalence class E of initial states which contains both S_1 and S_2, so $\omega_1 = \eta(S_1, \Omega_1)$ and $\omega_2 = \eta(S_2, \Omega_2)$ are both in J_E (Lemma 4.3-3). Denoting either of ω_1 and ω_2 by ω, if ω is not halted in J_E, then it is a proper prefix of some $\beta \in J_{S', \Omega'}$ where S' equals S and Ω' is a halted firing sequence starting in S'. Since $\Phi(\beta)$ is the reduction of Ω', $\Phi(\omega)$ is the reduction of a prefix θ of Ω' (Lemma 7.2-4). But $\Phi(\omega)$ is also the reduction of Ω (Lemma 4.3-3), so θ equals Ω. Therefore, $S' \cdot \theta$ equals $S \cdot \Omega$, and since no actor is enabled in $S \cdot \Omega$, no actor is enabled in $S' \cdot \theta$, so $\Omega' = \theta$, which is Ω. The number of tokens which appear at the nth firing in Ω equals the number which appear at the nth firing in Ω'. Since ω already contains an entry for each token which appears, β can have no more entries. I.e., ω cannot be a proper prefix of any computation in J_E, so it is halted in J_E (Lemma 7.3-1 below). Therefore, ω_1 and ω_2 are two halted computations in the same job, and so are equivalent computations.

By construction (Algorithm 4.3-1), there is a token of value v, (v,R), or (v,W) on an arc b in $S_1 \cdot \Omega_1$ ($S_2 \cdot \Omega_2$) iff there is an entry in ω_1 (ω_2) with value v whose transfer has a destination given by

Dst(Ex((d,j),0),1) if b is the number-j input arc of the

actor labelled d,

Dst(Ex(("OD",j),0),1) if b is the number-j program output arc

By equivalence, there is a one-to-one pointer correspondence F such that there is an entry f in ω_1 iff there is an entry g in ω_2 with the same transfer, and if V(f) is not a pointer, V(g) = V(f), otherwise

$V(g) = F(V(f))$. Therefore, there is a token on b in $S_1 \cdot \Omega_1$ iff there is one in $S_2 \cdot \Omega_2$, and either their values are the same non-pointer, or one is (p,R) or (p,W) and the other is $(F(p),R)$ or $(F(p),W)$.

For any actor d, the number of firings of d in Ω_1 equals the number of executions of d which have input entries in ω_1 which equals the number of executions of d which have input entries in ω_2 which equals the number of firings of d in Ω_2. If d is a gate and its control input arc is its number-j input arc, then the control input to the kth firing of d in Ω_1 (Ω_2) equals the value of $Ent(Ex(d,k),j)$ in ω_1 (ω_2), which is not a pointer; hence the kth firings of d in Ω_1 and Ω_2 have the same control input. The kth firing of d in Ω_1 (Ω_2) removes a token from an output arc of d' and is preceded by exactly k' firings of d' iff there is an entry in ω_1 (ω_2) with transfer $(Src(Ex(d',k'),i),Dst(Ex(d,k),j))$ for some i and j; therefore, the kth firing of d in Ω_1 removes a token from an output arc of d' and is preceded by k' firings of d' iff the kth firing of d in Ω_2 removes a token from an output arc of d' and is preceded by k' firings of d'. From these three facts, for each arc b which holds pointer-valued tokens in $S_1 \cdot \Omega_1$ and $S_2 \cdot \Omega_2$, either both are read pointers or both are write pointers (Lemma 7.2-5). Therefore, if b holds a token of value (p,R) ((p,W)) in $S_1 \cdot \Omega_1$, it holds a token of value $(F(p),R)$ ($(F(p),W)$) in $S_2 \cdot \Omega_2$.

The major task left is to prove that, letting the heap in $S_i \cdot \Omega_i$, $i=1,2$, be $U_i' = (N_i',\Pi_i',SM_i')$, there is a one-to-one mapping $I: N_1' \to N_2'$ such that, for every value (p,R) or (p,W) on an arc in $S_1 \cdot \Omega_1$, $U_2'.\Pi_2'(F(p)) \overset{I}{=} U_1'.\Pi_1'(p)$. Letting the heap in S_i be $U_i = (N_i,\Pi_i,SM_i)$, there is a one-to-one mapping $I_1: N_1 \to N_2$ such that, for every pointer p

on an arc in S_1, there is a pointer p' on that arc in S_2, and $U_2 \cdot \Pi_2(p') \overset{I_1}{=} U_1 \cdot \Pi_1(p)$. I is built on I_1 thusly:

$$I(n) = \begin{cases} I_1(n) & \text{if } n \in N_1 \\ \Pi_2'(F(p)) & \text{if } n \notin N_1 \text{ and } F(p) \text{ is defined} \end{cases}$$

where p is such that $\Pi_1'(p) = n$. In this way, $\Pi_2'(F(p)) = I(\Pi_1'(p))$, at least if p is not in dom Π_1.

Ideally, U_1' would be shown to be the heap determined by ω_1 from U_1; then the equality of reaches in the equivalent computations ω_1 and ω_2 would imply that each of U_1 and U_2 had been altered in the same way. Unfortunately, the heap determined by a computation is defined only for $EE(L_{BS}, S)$. Therefore, it is necessary to work with both the standard and the modified interpreters. Two useful results relating the two interpreters have already been derived: (1) For S_i' the initial standard state corresponding to S_i, $S_i' \cdot \Omega_i \mu S_i \cdot \Omega_i$, so in particular, the heap in $S_i' \cdot \Omega_i$ is also U_i'. (2) There is a halted firing sequence Ω_i' starting in S_i' which has Ω_i as a prefix such that $\beta_i = \eta(S_i', \Omega_i')$ is SOE-inclusive of ω_i.

The canonical computation $\alpha_i = \eta(S_i', \Omega_i)$ is a prefix of β_i. Any structure operation execution $e = Ex(d, k)$ is initiated in ω_i iff there are k firings of d in Ω_i iff e is initiated in α_i. For every j, if there is any entry $Ent(e, j)$ in ω_i, there is one with the same value in β_i, by SOE-inclusion; since e is initiated in α_i, that entry must be in α_i. Similarly, if e is initiated in α_i, then for any Assign, Update, or Delete execution A, $e \in R(A)$ in α_i iff $e \in R(A)$ in β_i iff $e \in R(A)$ in ω_i (Lemma 5.2-6). Therefore, by equivalence of ω_1 and ω_2, e has the same non-pointer inputs in α_1 and α_2, if it has a pointer input p in α_1, that pointer input in α_2

is $F(p)$, and $e \in R(A)$ in α_1 iff $e \in R(A)$ in α_2.

For NAR_1 the node activation record derived from Ω_1 and α_1, U_1' is the heap determined by α_1 from U_1 and NAR_1. For any particular pointer p and node n, $(p,n) \in \Pi_1' - \Pi_1$ iff $(p,n) \in \text{ran } NAR_1$ iff there is a Copy execution $C = Ex(d,k)$ such that $NAR_1(C) = (p,n)$, so that the kth firing of d in Ω_1 is $(d,(p,n))$ iff there is an entry in ω_1 with value p whose transfer has source $Src(C,j)$ for some j (Lemma 7.1-2). Since $Src(C,j)$ has value p in ω_1 iff it has value $F(p)$ in ω_2, $p \in \text{dom } \Pi_1' - \text{dom } \Pi_1$ iff $F(p) \in \text{dom } \Pi_2' - \text{dom } \Pi_2$. Furthermore, letting CC_1 be the Creating-Copy function corresponding to NAR_1, $CC_1(p)$ is defined and equal to C iff $\exists n: NAR_1(C) = (p,n)$ iff $Src(C,j)$ has value p in ω_1 iff $Src(C,j)$ has value $F(p)$ in ω_2 iff $\exists n': NAR_2(C) = (F(p),n')$ iff $CC_2(F(p))$ is defined and equal to C.

For any pointer p which is the value of a source s in ω_1, p is the value of s in β_1 (by SOE-inclusion) and $F(p)$ is the value of s in ω_2 (by equivalence); hence p is the value of s in β_1 and $F(p)$ is the value of s in β_2. If $s = Src(S,1)$ for a Select execution S, S is in no reach in β_1 iff S is in no reach in ω_1 (SOE-inclusion) iff S is in no reach in ω_2 (equivalence) iff S is in no reach in β_2. These are sufficient to prove that $(p,\beta_1)\rho(F(p),\beta_2)$, whence $p \in \text{dom } \Pi_1 \Rightarrow F(p) \in \text{dom } \Pi_2 \Rightarrow \Pi_2(F(p)) = I_1(\Pi_1(p))$, by Theorem 5.3-2. Thus, $\Pi_2'(F(p)) = I(\Pi_1'(p))$, whether or not p is in $\text{dom } \Pi_1$. Since I_1 is one-to-one, and Π_1, F, and Π_2 are all one-to-one, I is one-to-one.

Next it is proven that for any Assign, Update, or Delete execution A, and any pointer p, duration $D(A)$ extends to the end of $H_p^{\alpha_1}$ iff $D(A)$ extends to the end of $H_{F(p)}^{\alpha_2}$. $Ent_{\alpha_1}(A,1)$ is in $H_p^{\alpha_1}$ iff it has value p iff

$\text{Ent}_{\alpha_2}(A,1)$ has value $F(p)$ iff $\text{Ent}_{\alpha_2}(A,1)$ is in $H^{\alpha_2}_{F(p)}$. Assuming for simplicity that A is an Assign execution, $D(A)$ extends to the end of $H^{\alpha_1}_p$ iff either $\text{Ent}(A,1)$ is the last input entry to an Assign execution in $H^{\alpha_1}_p$, or there is no such entry and $CC_1(p)$ is defined and is in reach $R(A)$ in α_1. $\text{Ent}(A,1)$ is the last input entry to an Assign execution in $H^{\alpha_1}_p$ but not in $H^{\alpha_2}_{F(p)}$ $\Rightarrow$ there is an A' such that $\text{Ent}(A',1)$ follows $\text{Ent}(A,1)$ in $H^{\alpha_2}_{F(p)}$ $\Rightarrow$ A' $\epsilon R(A)$ in α_2 $\Rightarrow$ A' $\epsilon R(A)$ in α_1 $\Rightarrow$ since $\text{Ent}(A',1)$ is in the same access history, it must follow $\text{Ent}(A,1)$ in $H^{\alpha_1}_p$, a contradiction. There is no input entry to an Assign execution in $H^{\alpha_1}_p$, and $CC_1(p)$ is defined and is in $R(A)$ in α_1 $\Rightarrow$ there is no Assign input entry in $H^{\alpha_2}_{F(p)}$ and $CC_2(F(p))$ is defined and equal to $CC_1(p)$ $\Rightarrow$ $CC_2(F(p)) \epsilon R(A)$ in α_2. Similar reasoning applies if A is an Update or Delete execution.

For any $(p_1,n_1) \epsilon \Pi'_1$, the content of n_1 in the heap determined by α_1 from U_1 and NAR_1 (which is U'_1) depends on the inputs to executions whose durations extend to the end of $H^{\alpha_1}_{p_1}$ and on the content in U_1 of a certain node m_1. The node m_1, along with the pointer q_1 to it, are given by: if $(p_1,n_1) \epsilon \Pi_1$, then $(q_1,m_1) = (p_1,n_1)$; otherwise, q_1 is the unique pointer in dom Π_1 such that $p'_1 = V(\text{Ent}_{\alpha_1}(CC_1(p),1))$ is dynamically descended from q_1 in α_1. The set CP containing each pointer p such that p is the value of an input entry to a structure operation execution in α_1 or $CC_1(p)$ is defined is the set of pointers to nodes which are either accessed or created by firings in Ω_1. If p_1 is not in CP, then n_1 is not created in Ω_1, so $(p_1,n_1) \epsilon \Pi_1$, and $F(p_1)$ is not the input to a structure operation execution in α_2 and $CC_2(F(p))$ is not defined, so letting $p_2 = F(p_1)$, $(p_2,n_2) \epsilon \Pi_2$. Since there is then no input entry to a structure operation

execution in $H_{p_1}^{\alpha_1}$ and $CC_1(p_1)$ is not defined, no durations extend to the

end of $H_{p_1}^{\alpha_1}$, so $SM_1'(n_1) = SM_1(m_1)$. Since $(p_1,n_1) \in \Pi_1$, $m_1 = n_1$, so

$SM_1'(n_1) = SM_1(n_1)$.

For any $p_1 \in CP$ and $p_2 = F(p_1)$, p_1 is the value of an input entry to a

structure operation execution $\Rightarrow p_1$ is the value of a source in $\omega_1 \Rightarrow$

$(p_1,\beta_1)\rho(p_2,\beta_2)$. Then $(p_1,n_1) \in \Pi_1 \Rightarrow (p_2,n_2) \in \Pi_2 \Rightarrow q_1 = p_1 \Rightarrow$

$(q_1,\beta_1)\rho(q_2,\beta_2)$. $CC_1(p_1)$ is defined $\Rightarrow (p_1,n_1) \notin \Pi_1 \Rightarrow (p_2,n_2) \notin \Pi_2 \Rightarrow$

$DD_{\alpha_1}(q_1,p_1')$ and p_1' is the value of an input entry to a structure operation

execution in α_1 (namely $CC_1(p_1)$) $\Rightarrow DD_{\beta_1}(q_1,p_1')$ and $(p_1',\beta_1)\rho(p_2',\beta_2) \Rightarrow$

$(q_1,\beta_1)\rho(q_2,\beta_2)$ (by Theorem 5.3-2). Therefore, $p_1 \in CP \Rightarrow (q_1,\beta_1)\rho(q_2,\beta_2)$.

Recalling the significance of the ρ relation, q_1 and q_2 point to nodes

in the initial heaps U_1 and U_2 which have "equal" contents; i.e.,

$SM_2(m_2) = I_1(SM_1(m_1))$, so $SM_2(m_2) = I(SM_1(m_1))$.

For any $p_1 \in CP$, the same executions' durations extend to the end of

$H_{p_1}^{\alpha_1}$ and $H_{p_2}^{\alpha_2}$. If no Assign execution's duration extends to the ends of

those access histories, then the value in $SM_1'(n_1)$ is the value in $SM_1(m_1)$

which is the value in $SM_2(m_2)$ which is the value in $SM_2'(n_2)$. If there is

such an Assign execution A, then the value in $SM_1'(n_1)$ is $V(Ent_{\alpha_1}(A,2))$

which is $V(Ent_{\alpha_2}(A,2))$ which is the value in $SM_2'(n_2)$. Therefore, $SM_1'(n_1)$

and $SM_2'(n_2)$ have the same value. Similarly, there is an ordered pair

with selector s in it in $SM_1'(n_1)$ iff there is one in $SM_2'(n_2)$. If no

Update execution's duration extends to the ends of $H_{p_1}^{\alpha_1}$ and $H_{p_2}^{\alpha_2}$, then for

any node n', $(s,n') \in SM_1'(n_1)$ iff $(s,n') \in SM_1(m_1)$ iff $(s,I_1(n')) \in SM_2(m_2)$ iff

$(s,I(n')) \in SM_2'(n_2)$. If Update execution U's duration extends to the ends

of those access histories, then for any pointer r, $(s,\Pi_1'(r)) \in SM_1'(n_1)$ iff

$r = V(Ent_{\alpha_1}(U,3))$ iff $F(r) = V(Ent_{\alpha_2}(U,3))$ iff $(s,\Pi_2'(F(r))) \in SM_2'(n_2)$.
Therefore, since $\Pi_2'(F(r)) = I(\Pi_1'(r))$, $p_1 \in CP \Rightarrow SM_2'(n_2) = I(SM_1'(n_1))$.

Finally, for any pointer p such that (p,R) or (p,W) is the value of
a token on any arc b in $S_1 \cdot \Omega_1$, b holds a token of value $(F(p),R)$ or
$(F(p),W)$ in $S_2 \cdot \Omega_2$, and p is the value of an entry in $\omega_1 = \eta(S_1,\Omega_1)$, so
$\Pi_2'(F(p)) = I(\Pi_1'(p))$. For any pointer r such that $n = \Pi_1'(r)$ equals or is
reachable from $\Pi_1'(p)$ in U_1', $r \in CP \Rightarrow SM_2'(\Pi_2'(F(r))) = I(SM_1'(\Pi_1'(r))) \Rightarrow$
$SM_2'(I(n)) = I(SM_1'(n))$. If $r \notin CP$ and for no node n' on the path from $\Pi_1'(p)$
to n is the pointer to n' in CP, then none of those nodes is accessed or
created in the computation, so each of them has the same contents in U_1'
and U_1. Since $\Pi_1'(p)$ is not created, $p \in dom \Pi_1$, and $F(p) \in dom \Pi_2$; hence,
there must be a pointer q on an arc b in S_1 such that $\Pi_1'(p) = \Pi_1(p)$ equals
or is reachable from $\Pi_1(q)$ in U_1 (Theorem 5.3-2). Therefore, n equals or
is reachable from $\Pi_1(q)$ in U_1, so $SM_2(I_1(n)) = I_1(SM_1(n))$. Since
$SM_1'(n) = SM_1(n)$, $SM_2'(I(n)) = I(SM_1'(n))$.

If $r \notin CP$ but there are nodes on the path from $\Pi_1'(p)$ to n the pointers
to which are in CP, there is a last such node n'; i.e., the pointer p' to
n' is in CP, but the pointers to all nodes after n' on the path to n are
not in CP. Letting n" be the node immediately following n' on that path,
there is a selector s such that $(s,n") \in SM_1'(n')$. If p" is such that
$\Pi_1'(p") = n"$, there is an Update execution U with selector input s such
that $D(U)$ extends to the end of $H_{p'}^{\alpha_1} \Rightarrow p" = V(Ent_{\alpha_1}(U,3)) \Rightarrow p" \in CP$. Since
$p" \notin CP$, there is a pointer $q \in dom \Pi_1$ such that $(s,n") \in SM_1(\Pi_1(q))$. None of
the nodes on the path from n" to n is accessed, so their contents are
each the same in U_1 and U_1'. Since n is reachable from n" in U_1', it is
reachable from n", hence from $\Pi_1(q)$, in U_1. As before, there is a

pointer q' on an arc b in S_1 such that $\Pi_1(q)$ equals or is reachable from $\Pi_1(q')$ in U_1, so n is reachable from $\Pi_1(q')$ in U_1, sc $SM_2'(I(n)) = I(SM_1'(n))$.

In any case, then, for any node n equal to or reachable from $\Pi_1'(p)$ for any p on an arc b in $S_1 \cdot \Omega_1$, $SM_2'(I(n)) = I(SM_1'(n))$. Since b holds a pointer of value $F(p)$ in $S_2 \cdot \Omega_2$ and $\Pi_2'(F(p)) = I(\Pi_1'(p))$, $U_2' \cdot \Pi_2'(F(p)) \overset{I}{=} U_1' \cdot \Pi_1'(p)$; i.e., $\text{Match}((b, S_1 \cdot \Omega_1), I, (b, S_2 \cdot \Omega_2))$. Because $S_1 \cdot \Omega_1$ and $S_2 \cdot \Omega_2$ are modified states, completing the proof that they are equal requires establishing the following condition: Letting the pool component in $S_i \cdot \Omega_i$ be Q_i, for every label S of a Select operator in P, $\exists p_1: S \in Q_1(p_1) \Leftrightarrow \exists p_2: S \in Q_2(p_2) \Rightarrow U_2' \cdot \Pi_2'(p_2) \overset{I}{=} U_1' \cdot \Pi_1'(p_1)$.

For any Select operator S, there are the same number k of firings of S in Ω_1 and Ω_2, and thus there is a prefix $\theta_i \varphi_i$ of Ω_i in which φ_i is the kth firing of S. For any pointer p_i, $S \in Q_i(p_i)$ in $S_i \cdot \Omega_i$ iff S was placed in that pool at the last firing φ_i and remains there through to the end of Ω_i iff there are no tokens on S's number-1 output arcs at any point after θ_i iff there is no entry in ω_i with source $\text{Src}(\text{Ex}(S,k),1)$. There is no entry with that source in ω_1 iff there is no such entry in ω_2 (by equivalence). Therefore, $\exists p_1: S \in Q_1(p_1)$ in $S_1 \cdot \Omega_1$ iff $\exists p_2: S \in Q_2(p_2)$ in $S_2 \cdot \Omega_2$.

It has already been shown that if p_1 appears as the value of an entry in ω_1, then $U_2' \cdot \Pi_2'(p_2) \overset{I}{=} U_1' \cdot \Pi_1'(p_1)$. If not, all that can be said is that $\Pi_1'(p_1)$ is the s_1-successor of $\Pi_1'(p_1')$ in $S_1 \cdot \theta_1$, where p_1' and s_1 are the pointer and selector inputs to φ_1. Letting δ_1 be $\eta(S_1', \theta_1)$, the heap in $S_1' \cdot \theta_1$ (which is the heap in $S_1 \cdot \theta_1$) is the heap determined from U_1 by δ_1. If there is any Update execution whose duration extends to the end of

$H_{p_1'}^{\delta_1}$, then p_1 is the value of $Ent_{\delta_1}(U,3)$; i.e., p_1 is the value of an entry

in ω_1. Therefore, there is no Update execution U whose duration extends

to the end of $H_{p_1'}^{\delta_1}$. There is an entry $f_i = Ent(Ex(S,k),1)$ with value p_i'

which is in $\gamma_i = \eta(S',\theta_i\varphi_i)$ but is not in $\delta_i = \eta(S',\theta_i)$. D(U) extends to

the end of $H_{p_1'}^{\delta_1}$ iff $f_i \in D(U)$ in γ_i (Lemma 5.2-7) iff $Ex(S,k)\in R(U)$ in γ_i iff

$Ex(S,k)\in R(U)$ in ω_1. $Ex(S,k)\in R(U)$ in ω_1 iff $Ex(S,k)\in R(U)$ in ω_2, so there

is no Update execution whose duration extends to the end of history $H_{p_2'}^{\delta_2}$.

Therefore, there is a pair (q_1,m_1) in Π_1 such that $(s_1,\Pi_1'(p_1))$

is in $SM_i(m_i)$. Since $s_i = V(Ent_{\omega_i}(Ex(S,k),2))$, $s_1 = s_2$ by equivalence.

By a previous argument, $SM_2(m_2) = I_1(SM_1(m_1))$, so $\Pi_2'(p_2) = I(\Pi_1'(p_1))$.

Either $q_1 = p_1'$ or $DD_{\delta_1}(q_1,p_1')$; in any case, q_1 is the value of an entry

in ω_1. Since $\Pi_1(p_1)$ is reachable from m_1 in at least U_1, reasoning

similar to that given earlier for any node n equal to or reachable from

$\Pi_1(p_1)$ in U_1 shows that $SM_2'(I(n)) = I(SM_1'(n))$. Thus

$U_2'.\Pi_2'(p_2) \stackrel{I}{=} U_1'.\Pi_1'(p_1)$. Since $\exists p_1: S\in Q_1(p_1) \Leftrightarrow \exists p_2: S\in Q_2(p_2) \Rightarrow$

$U_2'.\Pi_2'(p_2) \stackrel{I}{=} U_1'.\Pi_1'(p_1)$, $S_2 \cdot \Omega_2$ equals $S_1 \cdot \Omega_1$, and P is functional.

__Lemma 7.3-1__ Let S be any initial modified state for an L_{BS} program P,

and let Ω be any halted firing sequence starting in S. Then for E the

equivalence class of initial states containing S, $\eta(S,\Omega)$ is in J_E and is

halted therein.

__Proof:__

(1) Let $\omega = \eta(S,\Omega)$. Then ω is in $J_{S,\Omega}$ Lemma 4.3-3

(2) ω is a prefix of ω, and $\Omega\in FS(S)$, so ω is in J_E (1)+Def. 4.3-3

Prove that ω is halted in J_E by contradiction. Assume

(3) ω is not halted in J_E

(4) There is another computation $\alpha \in J_E$ of which ω is a proper prefix

$$(3)+\text{Def. } 4.2-7$$

(5) α is a prefix of some $\beta \in J_{S', \Omega'}$, where $S' \in E$ and Ω' is a halted

firing sequence starting in S' (4)+Def. 4.3-3

(6) ω is a proper prefix of β, which is a permutation of $\eta(S', \Omega')$

$$(4)+(5)+\text{Def. } 4.3-5$$

(7) There is a prefix θ of Ω' whose reduction is $\Phi(\omega)$ (5)+(6)+Lemma 7.2-2

(8) $\Phi(\omega)$ is the reduction of Ω (1)+Lemma 4.3-3

(9) θ equals Ω (7)+(8)+Def. 2.4-5

(10) S' equals S (5)

(11) Since Ω is halted, no actor is enabled in $S \cdot \Omega$ Def. 2.3-1

(12) No actor is enabled in $S' \cdot \theta$ (10)+(9)+(11)+Cor. 7.1-2

(13) θ is halted, so $\theta = \Omega'$, so Ω' equals Ω (12)+(7)+(9)+Def. 2.3-1

(14) For every prefix $\Xi' \varphi'$ of Ω' in which φ' is the kth firing of an

actor d, there is a prefix $\Xi \varphi$ of Ω in which φ is the kth firing

of d and Ξ equals Ξ' (13)+Def. 2.4-5

(15) $S \cdot \Xi$ equals $S' \cdot \Xi'$ (16)+(10)+Thm. 7.1-2

(16) Every control arc in P has a <u>true</u> (<u>false</u>) token in $S' \cdot \Xi'$ iff it

has one in $S \cdot \Xi$ (15)+Defs. 7.1-2+3.4-1

(17) There is an entry f in $\eta(S', \Omega')$ which is not in $\omega = \eta(S, \Omega)$. Let the

destination in $T(f)$ be $Dst(Ex(d,k),j)$ (6)

(18) $d \notin DL \Rightarrow$ d is the label of an actor in P $\Rightarrow$ there is a prefix $\Xi' \varphi'$ of

Ω' in which φ' is the kth firing of the actor labelled d, and a

token is removed from d's number-j input arc in going from $S' \cdot \Xi'$

to $S' \cdot \Xi' \varphi'$ Def. 4.3-2+Alg. 4.3-1

(19) $\Rightarrow$ there is a prefix $\Xi\varphi$ of Ω in which φ is the kth firing of d, and

a token is removed from d's number-j input arc in going from

$S \cdot \Xi$ to $S \cdot \Xi\varphi$ (14)+(16)+Defs. 3.3-9+2.1-5

(20) $\Rightarrow$ there is an entry g in ω whose transfer has destination

Dst(Ex(d,k),j)   Alg. 4.3-1

(21) $S \cdot \Omega$ equals $S' \cdot \Omega'$ (10)+(13)+Thm. 7.1-2

(22) d$\in$DL $\Rightarrow$ there is an arc b, uniquely associated with d, which holds

a token in $S' \cdot \Omega'$, and k = 0 and j = 1   (17)+Alg. 4.3-1

(23) $\Rightarrow$ b holds a token in $S \cdot \Omega$ (21)+Defs. 7.1-2+3.4-1

(24) $\Rightarrow$ there is an entry g in ω whose transfer has destination

Dst(Ex(d,k),j)   (22)+Alg. 4.3-1

(25) There is an entry in ω, hence in $\eta(S',\Omega')$, whose transfer has the

same destination as T(f)   (18)+(20)+(22)+(24)+(6)

(26) Since f is the only entry in $\eta(S',\Omega')$ whose transfer has that

destination, that entry in ω is f; i.e. f is in ω (25)+Def. 4.2-6

Since (3) leads to a contradiction between (17) and (26), (3) is false;

i.e., ω is halted in J_E.

$\triangle$

__Theorem 7.3-1__ For any L_D program P, if the expansion of P from $EE(L_D,M)$

is determinate, then P running on the modified interpreter is functional.

__Proof__:

(1) Let S_1 and S_2 be any two equal modified states for P, and let Ω_1

and Ω_2 be any two halted firing sequences starting in S_1 and S_2

respectively. Then P is functional iff $S_2 \cdot \Omega_2$ equals $S_1 \cdot \Omega_1$

Def. 2.4-4

(2) There is a single equivalence class E of initial modified states

for P that contains both S_1 and S_2 Cor. 2.4-1

(3) Let $\omega_1 = \eta(S_1, \Omega_1)$ and $\omega_2 = \eta(S_2, \Omega_2)$. Then ω_1 and ω_2 are both

halted in J_E (2)+Lemma 7.3-1

(4) Letting the expansion of P be (Int, J), J_E is in J (2)+Def. 4.3-2

(5) ω_1 and ω_2 are equivalent computations under a one-to-one pointer

correspondence F   (3)+(4)+Def. 6.1-1

(6) The sets of transfers of the entries in ω_1 and ω_2 are identical

(5)+Def. 6.1-1

(7) Every arc in P is either one of an ordered set of input arcs of an

actor in P or one of an ordered set of program output arcs of P

Def. 2.1-1

(8) For every arc b in P, denote by AD(b) the destination

$Dst(Ex((d,j),0),1)$ if b is the number-j input arc of actor d

$Dst(Ex(("OD",i),0),1)$ if b is the number-i program output arc   (7)

(9) For every arc b in P, there is a token on b in $S_1 \cdot \Omega_1$ iff there is

an entry in ω_1 whose transfer has destination AD(b)

(8)+(3)+Alg. 4.3-1

(10) iff there is an entry in ω_2 whose transfer has destination AD(b) (6)

(11) iff there is a token on b in $S_2 \cdot \Omega_2$ (8)+(3)+Alg. 4.3-1

(12) Let f and g be two entries from ω_1 and ω_2 respectively, with the

same transfer. Then V(f) is not a pointer iff V(g) is not a

pointer, if those values are not pointers, then they are the same,

and if those values are pointers, then F(V(f)) is defined and

equal to V(g)   (5)+Def. 6.1-1

(13) For each arc b in P, there is a token on b with non-pointer value

v in $S_1 \cdot \Omega_1$ iff there is an entry in ω_1 with transfer t containing

destination AD(b), and a non-pointer value v   (8)+(3)+Alg. 4.3-1

(14) iff there is an entry in ω_2 with transfer t, containing destination

AD(b), and a non-pointer value v   (6)+(12)

(15) iff there is a token on b with non-pointer value v in $S_2 \cdot \Omega_2$

(8)+(3)+Alg. 4.3-1

(16) For every arc b in P, there is a token on b with value (p,R) or

(p,W), p a pointer, in $S_1 \cdot \Omega_1$ iff there is an entry in ω_1 with

transfer t containing destination AD(b) and pointer value p

(8)+(3)+Alg. 4.3-1

(17) iff there is an entry in ω_2 with transfer t, containing destination

AD(b), and pointer value F(p)   (5)+(12)

(18) iff there is a token on b with value (F(p),R) or (F(p),W), p a

pointer, in $S_2 \cdot \Omega_2$ (8)+(3)+Alg. 4.3-1

(19) Let Int = (St, I,IE). Then Int = Int(P), and d$\in$St-DL iff d is the

label of an actor in P   Def. 4.3-2

(20) Let e = Ex(d,k) be any execution in which d$\in$St-DL. Then e is

initiated in ω_1 (ω_2) iff there are In(I(d)) input entries to e in

ω_1 (ω_2)   Def. 4.2-6

(21) e is initiated in ω_1 iff e is initiated in ω_2 (20)+(6)+Def. 4.2-5

(22) For i=1,2, $\Phi(\omega_i)$ is the reduction of Ω_i and ω_i is in J_{S_i,Ω_i}

(1)+(3)+Lemma 4.3-3

(23) ω_i is a prefix of a causal permutation of $\eta(S_i,\Omega_i)$ (22)+Def. 4.3-5

(24) For any actor d in P, there are k firings of d in Ω_1 (or Ω_2) iff

Ex(d,k) is initiated in ω_1 (ω_2) and Ex(d,k+1) is not initiated

(19)+(1)+(22)+(23)+Thm. 4.3-2

(25) There are the same number of firings of d in Ω_1 and Ω_2 (24)+(21)

(26) If d is a gate, let j be such that d's control input arc is its number-j input arc. Then the value of the token removed from that arc by the kth firing of d in Ω_1 (Ω_2) equals V(Ent(Ex(d,k),j)) in ω_1 (ω_2) (3)+Alg. 4.3-1

(27) Since that value is not a pointer, V(Ent(Ex(d,k),j)) is the same in ω_1 and ω_2, so the kth firings of d in Ω_1 and Ω_2 remove control tokens of the same value (26)+(12)+Def. 2.2-1

(28) For any two actors d and d' and for any k, there is a k' such that the kth firing of d in Ω_1 (Ω_2) removes a token from an output arc of d' and exactly k' firings of d' precede it iff there is an entry in ω_1 (ω_2) with transfer (Src(Ex(d',k'),i),Dst(Ex(d,k),j)), for some i and j depending on the arc (3)+Alg. 4.3-1

(29) There is a k' such that if the kth firings of d in Ω_1 and Ω_2 remove tokens from output arcs of d', then those firings both are preceded by exactly k' firings of d' (28)+(6)

(30) For any arc b in P which holds tokens of pointer value in $S_1 \cdot \Omega_1$ and $S_2 \cdot \Omega_2$, either both are read pointers or both are write pointers

  (25)+(27)+(29)+Lemma 7.2-5

(31) b holds a token of value (p,R) ((p,W)), p a pointer, in $S_1 \cdot \Omega_1$ iff b holds a token of value (F(p),R) ((F(p),W)) in $S_2 \cdot \Omega_2$ (16)+(18)+(30)

Letting the heap in $S_1 \cdot \Omega_1$ be $U_i' = (N_i', \Pi_i', SM_i')$, i=1,2, the major task remaining is to prove that there is a single one-to-one mapping I: $N_1' \to N_2'$ such that, for every value (p,R) or (p,W), p a pointer, on an arc in $S_1 \cdot \Omega_1$, $U_2' \cdot \Pi_2'(F(p)) \overset{I}{=} U_1' \cdot \Pi_1'(p)$.

(32) Let S_1' and S_2' be the initial standard states for P corresponding to S_1 and S_2. Then, for i=1,2, Ω_i is a firing sequence starting

in S_i', and $S_i' \cdot \Omega_i \mu S_i' \cdot \Omega_i$ (1)+Thm. 7.1-1

(33) U_i' is the heap in $S_i' \cdot \Omega_i$ (32)+Def. 7.1-1

(34) For i=1,2, there is a halted firing sequence Ω_i' starting in S_i'

which has Ω_i as a prefix such that $\beta_i = \eta(S_i', \Omega_i')$ is SOE-inclusive

of ω_i (1)+(32)+(3)+Thm. 7.1-3

(35) $\alpha_i = \eta(S_i', \Omega_i)$ is a prefix of β_i (34)+Alg. 4.3-1

(36) ω_1, ω_2, α_1, α_2, β_1, and β_2 are all causal computations for Int(P)

         (1)+(3)+(32)+(34)+(35)+Lemma 4.3-2

(37) For i=1,2, for any structure operation execution e = Ex(d,k) and

any j, there is an entry Ent(e,j) in α_i iff d labels a structure

operation in P, there are k firings of d in Ω_i, and the kth

removes a token from d's number-j input arc   (35)+Alg. 4.3-1

(38) iff there is an entry Ent(e,j) in ω_i (3)+Def. 2.2-5+Alg. 4.3-1

(39) There is an entry f = Ent(e,j) in α_i $\Rightarrow$ there is exactly one entry

Ent(e,j) in β_i, and it has value V(f)   (35)+(36)+Def. 4.2-6

(40) $\wedge$ there is an entry g = Ent(e,j) in ω_i (37)+(38)

(41) $\Rightarrow$ there is an entry Ent(e,j) in β_i with value V(g)   (34)+Def. 5.2-8

(42) $\Rightarrow$ there is an entry Ent(e,j) in ω_i with value V(f)   (39)

(43) For any structure operation execution e and any j, there is an entry

Ent(e,j) in α_1 iff there is an entry Ent(e,j) in α_2

         (37)+(38)+(6)+Def. 4.2-6

(44) $\wedge$ $V(\text{Ent}_{\alpha_1}(e,j))$ is not a pointer iff $V(\text{Ent}_{\alpha_2}(e,j))$ is not a pointer,

if those values are not pointers, then they are the same, and if

they are pointers, then $V(\text{Ent}_{\alpha_2}(e,j)) = F(V(\text{Ent}_{\alpha_1}(e,j)))$

         (39)+(42)+(12)

(45) For i=1,2, for any structure operation execution e, e is initiated

in α_i iff there are $In(I(d))$ input entries to e in α_i iff there

are that many input entries to e in ω_i iff e is initiated in ω_i

$$(37)+(38)+Def. \ 4.2-6$$

(46) For any pointer p, p is the value of the output entries in β_i of a

Copy execution C $\Rightarrow$ the first such entry in β_i with value p is an

output entry of C $\hspace{2cm}$ (32)+(34)+Lemma 5.2-3

(47) For any structure operation execution e initiated in α_i, and any

Assign, Update, or Delete execution A, $e \in R(A)$ in β_i iff $e \in R(A)$ in

α_i only if A is initiated in α_i $\hspace{1cm}$ (36)+(35)+(46)+Lemma 5.2-6

(48) $\wedge$ e is initiated in ω_i $\hspace{4cm}$ (45)

(49) $\Rightarrow e \in R(A)$ in ω_i iff $e \in R(A)$ in β_i $\hspace{1cm}$ (36)+(34)+(46)+Lemma 5.2-6

(50) For any structure operation execution e initiated in both α_1 and

α_2, and any Assign, Update, or Delete execution A, $e \in R(A)$ in α_1

iff $e \in R(A)$ in ω_1 iff $e \in R(A)$ in ω_2 iff $e \in R(A)$ in α_2

$$(47)+(49)+(5)+Def. \ 6.1-1$$

(51) For i=1,2, let the heap in S_i' be $U_i = (N_i, \Pi_i, SM_i)$. Let NAR_i be the

node activation record derived from Ω_i and α_i. Then the heap in

$S_i' \cdot \Omega_i$, U_i', is the heap determined by α_i from U_i and NAR_i

$$(32)+(35)+(33)+Thm. \ 5.2-1$$

(52) $\Pi_i \subseteq \Pi_i'$, and for all (p,n), $(p,n) \in \Pi_i' - \Pi_i$ iff $(p,n) \in ran \ NAR_i$

$$(51)+Def. \ 5.2-7$$

(53) iff there is a Copy execution C such that $NAR_i(C) = (p,n)$ iff

C = Ex(d,k) where the kth firing of d in Ω_i is (d,(p,n))

$$(51)+Defs. \ 5.2-1+5.2-4$$

(54) iff there is an entry in ω_i with value p whose transfer has source

Src(C,j) for some j $\hspace{3cm}$ (1)+Lemma 7.1-2

(55) For all $p \in \text{dom } \Pi_1' - \text{dom } \Pi_1$, p is the value of an entry in ω_1

$$(52)+(54)+(12)$$

(56) For every pointer p, $p \in \text{dom } \Pi_1' - \text{dom } \Pi_1$ iff there is a Copy execution

C and an n such that $\text{NAR}_1(C) = (p,n)$ iff there is an entry in ω_1

whose transfer has source $\text{Src}(C,j)$ for some j $\qquad (52)+(53)+(54)$

(57) iff there is an entry in ω_2 whose transfer has source $\text{Src}(C,j)$ for

some Copy execution C and some j and whose value is $F(p)$ $\quad (6)+(12)$

(58) iff there is a Copy execution C and an n' such that $\text{NAR}_2(C)$ is

$(F(p),n')$ iff $F(p) \in \text{dom } \Pi_2' - \text{dom } \Pi_2$ $\qquad (52)+(53)+(54)$

(59) Let CC_1 be the Creating-Copy function corresponding to NAR_1. Then

for any pointer p and Copy execution C, $\text{CC}_1(p)$ is defined and equal

to C iff $\exists n: \text{NAR}_1(C) = (p,n)$ iff $\exists n': \text{NAR}_2(C) = (F(p),n')$ iff

$\text{CC}_2(F(p))$ is defined and equal to C $\qquad (56)+(58)+\text{Def. } 5.2\text{-}5$

(60) For any pointer p, p appears as the value of an entry in $\omega_1 \Rightarrow$

there is an entry in ω_1 with value p whose transfer has a source s

$\Rightarrow$ there is an entry in β_1 with value p whose transfer has source s

$$(34)+\text{Def. } 5.2\text{-}8$$

(61) $\wedge$ $F(p)$ is defined and there is an entry in ω_2 with value $F(p)$ whose

transfer has source s $\qquad (6)+(12)$

(62) $\Rightarrow$ there is an entry in β_2 with value $F(p)$ whose transfer has source

s $\qquad (34)+\text{Def. } 5.2\text{-}8$

Prove by contradiction that for any pointer p which appears as the value

of an entry in ω_1, $(p,\beta_1)\rho(F(p),\beta_2)$. Assume this is false; i.e.,

(63) there is a prefix γf of ω_1 such that, for every pointer q which

appears as the value of an entry in γ, $(q,\beta_1)\rho(F(q),\beta_2)$, but for

$p = V(f)$, $(p,\beta_1)\not\rho(F(p),\beta_2)$

(64) f is the first entry in ω_1 with value p $\hfill$ (63)

(65) Let e be the execution of which f is an output entry. Then there

is an entry in ω_1 with value p whose transfer has source

s = Src(e,j) for some j $\hfill$ (63)+Def. 4.2-5

(66) p is the value of source s in β_1, F(p) is defined, and F(p) is

the value of s in β_2 $\hfill$ (65)+(60)+(61)+(62)+Def. 4.2-6

(67) e either is in IE, is a Copy execution, or is a Select execution

which is in no reach in ω_1 $\hfill$ (36)+(64)+Lemma 5.3-8

(68) e is initiated in γ $\hfill$ (63)+(65)+(36)+Def. 4.2-7

(69) e is initiated in ω_1, hence in ω_2 $\hfill$ (45)+(6)+Def. 4.2-6

(70) $e \in IE \Rightarrow (p,\beta_1)\rho(F(p),\beta_2)$ $\hfill$ (65)+(66)+Def. 5.1-10

(71) e is a Select execution which is in no reach in ω_1 $\Rightarrow$ e is a Select

execution which is in no reach in ω_2 $\hfill$ (5)+Def. 6.1-1

(72) $\Rightarrow$ e is in no reach in β_1 or β_2 $\hfill$ (69)+(36)+(34)+(46)+Lemma 5.2-6

(73) e is a Select execution $\Rightarrow$ there is an Ent(e,1) and an Ent(e,2) in

ω_1, hence in ω_2, and the former's values are pointers, while the

latter's are not pointers $\hfill$ (69)+(6)+Def. 4.2-6+Const. 5.1-1

(74) $\Rightarrow$ there are entries Ent(e,1) and Ent(e,2) in β_1 and β_2, and for

j=1,2, $V(Ent_{\beta_1}(e,j)) = V(Ent_{\omega_1}(e,j))$ $\hfill$ (34)+Def. 5.2-8

(75) $\Rightarrow V(Ent_{\beta_1}(e,2)) = V(Ent_{\beta_2}(e,2))$ and $V(Ent_{\beta_2}(e,1)) = F(V(Ent_{\beta_1}(e,1)))$

$\hfill$ (12)

(76) $\wedge V(Ent_{\omega_1}(e,1)) = V(Ent_{\beta_1}(e,1))$ is the value of an entry in γ

$\hfill$ (73)+(68)+Def. 4.2-6

(77) $\Rightarrow (V(Ent_{\beta_1}(e,1)),\beta_1)\rho(V(Ent_{\beta_2}(e,1)),\beta_2)$ $\hfill$ (63)+(75)

(78) e is a Select execution which is in no reach in ω_1 $\Rightarrow$

$(p,\beta_1)\rho(F(p),\beta_2)$ $\hfill$ (66)+(71)+(72)+(73)+(75)+(77)+Def. 5.1-10

(79) e is a Copy execution   (63)+(67)+(70)+(78)

(80) There is an entry $Ent(e,1)$ in γ of pointer value q

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ (68)+Def. 4.2-6+Const. 5.1-1

(81) $(q,\beta_1)\rho(F(q),\beta_2)$, there are entries $Ent(e,1)$ in β_1 and β_2,

$\qquad V(Ent_{\beta_1}(e,1)) = q$ and $V(Ent_{\beta_2}(e,1)) = F(q)$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ (80)+(63)+(6)+(12)+(34)+Def. 5.2-8

(82) $DD_{\beta_1}(q,p)$ and $DD_{\beta_2}(F(q),F(p))$ $\qquad\qquad$ (66)+(81)+Def. 5.1-9

(83) $q \neq p$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ (80)+(64)

(84) $(p,\beta_1)\rho(F(q),\beta_2)$ $\qquad\qquad\qquad\qquad$ (83)+(82)+(81)+Def. 5.1-10

(85) $(p,\beta_1)\rho(F(p),\beta_2)$ $\qquad\qquad\qquad\qquad$ (83)+(82)+(85)+Def. 5.1-10

Since (63) leads to a contradiction with (85), (63) is false. I.e.,

(86) For every pointer p which is the value of an entry in ω_1,

$\qquad (p,\beta_1)\rho(F(p),\beta_2)$

(87) For any pointer p, p is the value of an input entry to a structure

$\qquad$ operation execution in $\alpha_1 \Rightarrow$ p is the value of an entry in ω_1

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ (37)+(39)+(42)

(88) $\Rightarrow F(p)$ is defined and $(p,\beta_1)\rho(F(p),\beta_2)$ $\qquad$ (12)+(86)

(89) S_1' and S_2' are equal initial standard states $\qquad$ (1)+(32)+Thm. 7.1-2

(90) There is a single one-to-one mapping $I_1: N_1 \to N_2$ such that, for each

$\qquad$ arc b in P, $Match((b,S_2'), I_1, (b,S_1'))$ $\qquad\qquad$ (51)+(89)+Def. 2.4-3

(91) Define a mapping $I: N_1' \to N_2'$ by

$$I(n) = \begin{cases} I_1(n) & \text{if } n \in N_1 \\ \Pi_2'(F(p)) & \text{if } n \notin N_1 \text{ and } F(p) \text{ is defined} \end{cases}$$

$\qquad$ where p is such that $\Pi_1'(p) = n$

(92) For all $p_1 \in \text{dom } \Pi_1'$ and $p_2 \in \text{dom } \Pi_2'$, $p_2 = F(p_1)$ and p_1 is the value of

$\qquad$ any entry in ω_1 or any input entry to a structure operation

execution in $\alpha_1 \Rightarrow [p_1 \in \text{dom } \Pi_1 \Rightarrow p_2 \in \text{dom } \Pi_2$ (56)+(58)

(93) $\Rightarrow \Pi_2(p_2) = I(\Pi_1(p_1))]$ (87)+(88)+(51)+(34)+(36)+Thm. 5.3-2

(94) $\wedge [p_1 \notin \text{dom } \Pi_1 \Rightarrow \Pi_1'(p_1) \notin N_1$ (51)+Def. 2.2-1

(95) $\Rightarrow \Pi_2'(p_2) = I(\Pi_1'(p_1))] \Rightarrow \Pi_2'(p_2) = I(\Pi_1'(p_1))$ (91)

Next prove that I is one-to-one

(96) Let n be any node in N_1'. $n \in N_1 \Rightarrow$ there is a unique $n' \in N_2$ such that

$n' = I_1(n) = I(n)$ (90)+(91)

(97) For i=1,2, NAR_i is compatible with α_i, and ran NAR_i is consistent

with U_i (32)+(35)+(51)+Lemma 5.2-2

(98) $n \notin N_1 \Rightarrow \exists p: (p,n) \in \Pi_1' - \Pi_1 = \text{ran } NAR_1$ (51)+(52)+Def. 5.2-7

(99) $\Rightarrow$ there is a unique p such that $\Pi_1'(p) = n \wedge p \in \text{dom } \Pi_1' - \text{dom } \Pi_1$

(97)+Def. 5.2-3

(100) $\Rightarrow F(p)$ is defined, is unique, and is in dom $\Pi_2' - \text{dom } \Pi_2 = \text{ran } NAR_2$

(55)+(5)+(56)+(58)+(57)

(101) $\Rightarrow$ there is a unique $n' = \Pi_2'(F(p)) = I(n)$ (97)+(91)+Defs. 5.2-3+5.2-7

(102) Let n_1 and n_2 be any two nodes in N_1'. n_1 is in N_1 and n_2 is not

$\Rightarrow I(n_1) = I_1(n_1)$, which is in N_2, and $I(n_2) = \Pi_2'(F(p))$, where p

is such that $\Pi_1'(p) = n_2 \notin N_1$ (90)+(98)+(101)

(103) $\Rightarrow p \in \text{dom } \Pi_1' - \text{dom } \Pi_1 \Rightarrow F(p) \in \text{dom } \Pi_2' - \text{dom } \Pi_2$ (51)+(56)+(58)+Def. 2.2-1

(104) $\Rightarrow (F(p), \Pi_2'(F(p))) \in \Pi_2' - \Pi_2 = \text{ran } NAR_2$ (51)+Def. 2.2-1

(105) $\Rightarrow I(n_2) = \Pi_2'(F(p))$ is not in N_2 (97)+(51)+(91)+Def. 5.2-3

(106) $I(n_1) = I(n_2) \Rightarrow$ either n_1 and n_2 are both in N_1 or they are both

not in N_1 (102)+(105)

(107) n_1 and n_2 are both in $N_1 \Rightarrow I(n_2) = I(n_1)$ iff $I_1(n_2) = I_1(n_1)$

(91)+(90)

(108) n_1 and n_2 are both not in $N_1 \Rightarrow I(n_2) = I(n_1) \Rightarrow$

$$\Pi_2'(F(p_2)) = \Pi_2'(F(p_1)) \text{ where, for } i=1,2, \Pi_1'(p_1) = n_1 \Rightarrow F(p_1) = F(p_2)$$

$$(51)+\text{Def. 2.2-1}$$

$(109) \Rightarrow p_1 = p_2 \Rightarrow n_1 = n_2$ $(5)+(51)+\text{Def. 2.2-1}$

(110) I is one-to-one $(96)+(98)+(101)+(106)+(107)+(108)+(109)$

Next prove that for any Assign, Update, or Delete execution A and for any pointer p, duration D(A) extends to the end of $H_p^{\alpha_1}$ iff D(A) extends to the end of $H_{F(p)}^{\alpha_2}$.

(111) Let A be any Assign execution. Then D(A) extends to the end of $H_p^{\alpha_1}$ iff either

$(111a)$ $\text{Ent}_{\alpha_1}(A,1)$ is the last number-1 input entry to an Assign execution in that access history, or

$(111b)$ there is no such entry in $H_p^{\alpha_1}$, and $CC_1(p)$ is defined and is in reach R(A) in α_1 $(97)+(51)+\text{Def. 5.2-7}$

(112) For any structure operation execution e, e is initiated in α_1 iff e is initiated in ω_1 iff e is initiated in ω_2 iff e is initiated in α_2 $(45)+(6)+\text{Def. 4.2-6}$

(113) For any structure operation execution e, $\text{Ent}_{\alpha_1}(e,1)$ is in $H_p^{\alpha_1}$ iff its value is p and e is initiated in α_1 Def. 5.1-4

(114) iff e is initiated in α_2 and $V(\text{Ent}_{\alpha_2}(e,1)) = F(p)$ $(112)+(43)+(44)$

(115) iff $\text{Ent}_{\alpha_2}(e,1)$ is in $H_{F(p)}^{\alpha_2}$ Def. 5.1-4

(116) $(111a)$ iff $\text{Ent}_{\alpha_1}(A,1)$ is in $H_p^{\alpha_1}$ and there is no Assign execution A' such that $\text{Ent}_{\alpha_1}(A',1)$ follows $\text{Ent}_{\alpha_1}(A,1)$ in $H_p^{\alpha_1}$ with no intervening number-1 input entry to an Assign execution iff $\text{Ent}_{\alpha_1}(A,1)$ is in $H_p^{\alpha_1}$ and there is no Assign execution A' such that $\text{Ent}_{\alpha_1}(A',1)$ is in history $H_p^{\alpha_1}$ and is in D(A) in α_1 Def. 5.1-5

(117) iff $\text{Ent}_{\alpha_1}(A,1)$ is in $H_p^{\alpha_1}$ and there is no Assign execution A' such that $\text{Ent}_{\alpha_1}(A',1)$ is in that history and A' is in R(A) in α_1

<div align="right">Def. 5.1-6</div>

(118) iff $\text{Ent}_{\alpha_2}(A,1)$ is in $H_{F(p)}^{\alpha_2}$ and there is no Assign execution A' such that $\text{Ent}_{\alpha_2}(A',1)$ is in $H_{F(p)}^{\alpha_2}$ and A' is in R(A) in α_2

<div align="right">(113)+(115)+(114)+(50)</div>

(119) iff $\text{Ent}_{\alpha_2}(A,1)$ is the last number-1 input entry to an Assign execution in $H_{F(p)}^{\alpha_2}$

<div align="right">Defs. 5.1-5+5.1-6</div>

(120) For any pointer p, one of $CC_1(p)$ or $CC_2(F(p))$ is defined $\Rightarrow$ both are defined and are equal to the same Copy execution C

<div align="right">(59)</div>

(121) $\Rightarrow$ for i=1,2, $NAR_i(C)$ is defined

<div align="right">(59)+Def. 5.2-5</div>

(122) $\Rightarrow$ C is initiated in both α_1 and α_2

<div align="right">(51)+Def. 5.2-4</div>

(123) $\Rightarrow CC_1(p) \in R(A)$ in α_1 iff $CC_2(F(p)) \in R(A)$ in α_2

<div align="right">(50)</div>

(124) (111b) iff there is no number-1 input entry to an Assign execution in $H_{F(p)}^{\alpha_2}$

<div align="right">(113)+(115)</div>

(125) and $CC_2(F(p))$ is defined and is in R(A) in α_2

<div align="right">(120)+(123)</div>

(126) (111a) $\Rightarrow$ A is initiated in α_1

<div align="right">(113)</div>

(127) (111b) $\Rightarrow$ A is initiated in α_1

<div align="right">(36)+Lemma 5.3-8</div>

(128) D(A) extends to the end of $H_{F(p)}^{\alpha_2}$ iff D(A) extends to the end of $H_p^{\alpha_1}$ only if A is initiated in α_1

<div align="right">(111)+(116)+(119)+(124)+(125)+(126)+(127)+Def. 5.2-6</div>

(129) Let U be any Update or Delete execution initiated in α_1. Then

$$V(\text{Ent}_{\alpha_2}(U,2)) = V(\text{Ent}_{\alpha_1}(U,2))$$

<div align="right">(43)+(44)</div>

Replacing "Assign execution A" with "Update or Delete execution U with V(Ent(U,2)) = s" in (111) through (128) yields a proof of

(130) D(U) extends to the end of $H_p^{\alpha_1}$ iff D(U) extends to the end of $H_{F(p)}^{\alpha_2}$

only if U is initiated in α_1 $\hspace{2cm}$ (129)

(131) For any pointer p, $CC_1(p)$ is defined $\Rightarrow$ there is a Copy execution

C and a node n such that $NAR_1(C) = (p,n)$ $\Rightarrow$ there is an entry in

ω_1 with value p $\hspace{4cm}$ (59)+(53)+(54)

(132) $\wedge$ $(p,n) \in$ ran $NAR_1 = \Pi_1' - \Pi_1$ $\hspace{2cm}$ (52)+Def. 5.2-1

(133) $\Rightarrow F(p)$ is defined $\hspace{4cm}$ (12)

(134) $\wedge$ $F(p) \in$ dom Π_2' - dom Π_2 $\hspace{3cm}$ (56)+(58)

(135) For any pointer p, p is the value of an input entry of a structure

operation execution in α_1 $\Rightarrow$ $F(p)$ is the value of an entry in α_2

$\hspace{8cm}$ (43)+(44)

(136) $\Rightarrow$ there is a prefix θ of Ω_2 such that a token of value $F(p)$ is on

an arc in $S_2' \cdot \theta$ $\hspace{3cm}$ (34)+(35)+Alg. 4.3-1

(137) $\Rightarrow F(p)$ is in dom Π in $S_2' \cdot \theta$ $\hspace{1.5cm}$ (32)+Def. 2.3-1+Thm. 2.2-1

(138) $\Rightarrow F(p)$ is in dom Π_2' $\hspace{3cm}$ (33)+Def. 2.2-5

(139) Let CP be the set of pointers $\{p \mid p \in$ dom Π_1' and p is the value of

an input entry to a structure operation execution in α_1 or $CC_1(p)$

is defined$\}$. For any $p \in CP$, $F(p)$ is defined and is in dom Π_2'

$\hspace{3cm}$ (87)+(88)+(131)+(134)+(135)+(138)

Now prove that for any $p_1 \in CP$, $SM_2'(\Pi_2'(F(p_1))) = I(SM_1'(\Pi_1'(p_1)))$

(140) Let p_2 be $F(p_1)$ and let n_i be such that $(p_i,n_i) \in \Pi_i'$, i=1,2. Let

(q_i,m_i) be such that

If $(p_i,n_i) \in \Pi_i$, then $(q_i,m_i) = (p_i,n_i)$,

otherwise, (q_i,m_i) is the unique pair in Π_i such that, for

$p_i' = V(Ent_{\alpha_i}(CC_1(p),1))$, $DD_{\alpha_i}(q_i,p_i')$

$\hspace{3cm}$ (139)+(32)+(51)+(34)+(35)+Lemma 5.2-4

(141) $(p_1,n_1) \in \Pi_1$ $\Rightarrow$ $CC_1(p_1)$ is not defined $\wedge$ $p_1 = q_1$ $\Rightarrow$ p_1 is the value

of an input entry to a structure operation execution in α_1

$$(131)+(132)+(140)+(139)$$

(142) $\wedge \ (p_2,n_2) \in \Pi_2 \rightarrow q_2 = p_2$ \hfill $(139)+(56)+(58)+(140)$

(143) $\rightarrow (p_1,\beta_1)\rho(p_2,\beta_2) \rightarrow (q_1,\beta_1)\rho(q_2,\beta_2)$ \hfill $(87)+(88)$

(144) $(p_1,n_1)\notin \Pi_1 \rightarrow (p_2,n_2)\notin \Pi_2 \rightarrow DD_{\alpha_1}(q_1,p_1') \text{ and } DD_{\alpha_2}(q_2,p_2')$

$$(139)+(56)+(58)+(140)$$

(145) $\rightarrow$ since dynamic descendancy depends only on the entries in a

computation, $DD_{\beta_1}(q_1,p_1') \text{ and } DD_{\beta_2}(q_2,p_2')$ \hfill $(35)+$Def. 5.1-9

(146) $(p_1,n_1)\notin \Pi_1$ and p_1 is the value of an entry in $\alpha_1 \rightarrow CC_1(p_1)$ is

defined and $Ent_{\alpha_1}(CC_1(p_1),1)$ is in α_1

$$(32)+(34)+(35)+(51)+(59)+\text{Lemma 5.2-3}$$

(147) $CC_1(p_1)$ is defined $\rightarrow NAR_1(CC_1(p_1))$ is defined $\rightarrow CC_1(p_1)$ is

initiated in α_1 \hfill Defs. 5.2-5+5.2-4

(148) $\rightarrow Ent_{\alpha_1}(CC_1(p_1),1)$ is in α_1 \hfill Defs. 5.1-1+4.2-6

(149) $(p_1,n_1)\notin \Pi_1 \rightarrow p_1'$ is the value of an input entry to a structure

operation execution in α_1 and $CC_1(p_1)$ is defined \hfill $(139)+(146)-(148)$

(150) $\rightarrow (p_1',\beta_1)\rho(F(p_1'),\beta_2)$ \hfill $(87)+(88)$

(151) $\wedge \ p_2' = V(Ent_{\alpha_2}(CC_2(p_2),1)) = F(V(Ent_{\alpha_1}(CC_1(p_1),1))) = F(p_1')$

$$(140)+(59)+(43)+(44)$$

(152) $\rightarrow (q_1,\beta_1)\rho(q_2,\beta_2)$ \hfill $(89)+(90)+(51)+(35)+(36)+(144)+(145)+$Thm. 5.3-2

(153) $(q_1,\beta_1)\rho(q_2,\beta_2)$ \hfill $(141)+(143)+(149)+(152)$

(154) $SM_2(m_2) = I_1(SM_1(m_1))$

$$(89)+(90)+(51)+(32)+(35)+(36)+(140)+(153)+\text{Thm. 5.3-2}$$

(155) There is an Assign execution A such that D(A) extends to the end of

$H_{p_1}^{\alpha_1} \rightarrow$ the value in $SM_1'(n_1)$ equals $V(Ent_{\alpha_1}(A,2))$ \hfill $(51)+$Def. 5.2-7

(156) $\wedge$ D(A) extends to the end of $H_{P_2}^{\alpha_2} \wedge$ A is initiated in α_1 (140)+(128)

(157) $\Rightarrow$ the value in $SM_1'(n_1)$ equals $V(Ent_{\alpha_2}(A,2))$ (43)+(44)

(158) $\wedge$ the value in $SM_2'(n_2)$ equals $V(Ent_{\alpha_2}(A,2))$ (51)+Def. 5.2-7

(159) There is no Assign execution whose duration extends to the end of

$H_{P_1}^{\alpha_1} \Rightarrow$ the value in $SM_1'(n_1)$ equals the value in $SM_1(m_1)$

$$(51)+(140)+\text{Def. } 5.2\text{-}7$$

(160) $\wedge$ there is no Assign execution whose duration extends to the end

of $H_{P_2}^{\alpha_2}$ (140)+(128)

(161) $\Rightarrow$ the value in $SM_1'(n_1)$ equals the value in $SM_2(m_2)$ (154)+Def. 2.4-1

(162) $\wedge$ the value in $SM_2'(n_2)$ equals the value in $SM_2(m_2)$

$$(51)+(140)+\text{Def. } 5.2\text{-}7$$

(163) The value in $SM_2'(n_2)$ equals the value in $SM_1'(n_1)$

$$(155)+(157)+(158)+(159)+(161)+(162)$$

(164) For any selector s, there is an Update or Delete execution U such

that $V(Ent_{\alpha_1}(U,2)) = s$ and D(U) extends to the end of $H_{P_1}^{\alpha_1} \Rightarrow$ there

is an ordered pair with s in it in $SM_1'(n_1)$ iff U is an Update

$$(51)+\text{Def. } 5.2\text{-}7$$

(165) $\wedge$ D(U) extends to the end of $H_{P_2}^{\alpha_2} \wedge$ U is initiated in α_2 (140)+(130)

(166) $\Rightarrow V(Ent_{\alpha_2}(U,2)) = s$ (43)+(44)

(167) $\Rightarrow$ there is an ordered pair with s in it in $SM_2'(n_2)$ iff U is an

Update   (51)+Def. 5.2-7

(168) $\Rightarrow$ there is an ordered pair with s in it in $SM_2'(n_2)$ iff there is an

ordered pair with s in it in $SM_1'(n_1)$ (164)

(169) For any selector s, there is an ordered pair with s in it in

$SM_1(m_1)$ iff there is an ordered pair with s in it in $SM_2(m_2)$

$$(154)+\text{Def. } 2.4\text{-}1$$

(170) For any selector s, there is no Update or Delete execution U such that $V(\text{Ent}_{\alpha_1}(U,2)) = s$ and $D(U)$ extends to the end of $H_{P_1}^{\alpha_1} \Rightarrow$ there is an ordered pair with s in it in $SM_1'(n_1)$ iff the same ordered pair is in $SM_1(m_1)$

(51)+(140)+Def. 5.2-7

(171) $\wedge$ there is no Update or Delete execution U such that $V(\text{Ent}_{\alpha_2}(U,2))$ is s and $D(U)$ extends to the end of $H_{P_2}^{\alpha_2}$

(130)+(43)+(44)

(172) $\Rightarrow$ there is an ordered pair with s in it in $SM_2'(n_2)$ iff the same ordered pair is in $SM_2(m_2)$

(51)+(140)+Def. 5.2-7

(173) $\Rightarrow$ there is an ordered pair with s in it in $SM_2'(n_2)$ iff there is an ordered pair with s in it in $SM_1'(n_1)$

(170)+(169)

(174) For every selector s, there is an ordered pair with s in it in $SM_2'(n_2)$ iff there is an ordered pair with s in it in $SM_1'(n_1)$

(164)+(168)+(170)+(173)

(175) Let s be any selector such that there is a pair $(s,\Pi_1'(r_1))$ in $SM_1'(n_1)$. Then there is a pair $(s,\Pi_2'(r_2))$ in $SM_2'(n_2)$

(174)

(176) For i=1,2, $\Pi_i'(r_i) \in N_i'$, so $r_i \in \text{dom } \Pi_i'$

(32)+(33)+Thm. 2.2-1

(177) Either there is an Update execution U such that $D(U)$ extends to the ends of $H_{P_1}^{\alpha_1}$ and $H_{P_2}^{\alpha_2}$ and U is initiated in α_1, or $(s,\Pi_1'(r_1))$ is in $SM_1(m_1)$

(175)+(164)+(165)+(170)+(172)

(178) There is an Update execution U such that $D(U)$ extends to the ends of both $H_{P_1}^{\alpha_1}$ and $H_{P_2}^{\alpha_2}$ and U is initiated in $\alpha_1 \Rightarrow r_1 = V(\text{Ent}_{\alpha_1}(U,3))$

(51)+Def. 5.2-7

(179) $\Rightarrow r_2 = F(r_1)$

(43)+(44)

(180) $\Rightarrow \Pi_2'(r_2) = I(\Pi_1'(r_1))$

(176)+(92)+(95)

(181) $(s,\Pi_1'(r_1)) \in SM_1(m_1) \Rightarrow \Pi_2'(r_2) = I_1(\Pi_1'(r_1))$

(154)+Def. 2.4-1

(182) $\wedge \Pi_1'(r_1) \in N_1$

(51)+Def. 2.2-1

(183) $\Rightarrow \Pi_2'(r_2) = I(\Pi_1'(r_1))$ (91)

(184) For any selector s, there is a pair $(s,\Pi_2'(r_2))$ in $SM_2'(n_2)$ iff there

is a pair $(s,\Pi_1'(r_1))$ in $SM_1'(n_1)$ and $\Pi_2'(r_2) = I(\Pi_1'(r_1))$

(174)+(175)+(177)+(178)+(180)+(181)+(183)

(185) For every pointer $p \in CP$, $SM_2'(\Pi_2'(F(p))) = I(SM_1'(\Pi_1'(p)))$, so

$SM_2'(I(\Pi_2'(p))) = I(SM_1'(\Pi_1'(p)))$ (163)+(184)+(140)+(92)+(95)+Def. 2.4-1

(186) Let p_1 be any pointer in dom Π_1' but not in CP and let n_1 be $\Pi_1'(p_1)$.

Let n_2 be $I(n_1)$, and let p_2 be such that $\Pi_2'(p_2) = n_2$. p_2 is the

value of an input entry to a structure operation execution in α_2

$\Rightarrow p_2 = F(p')$ where p' is the value of an input entry to a

structure operation execution in α_1 (43)+(44)

(187) $\Rightarrow \Pi_2'(p_2) = I(\Pi_1'(p'))$ (92)+(95)

(188) $\Rightarrow$ since I and Π_1' are one-to-one, $p_1 = p'$, which is in CP

(110)+(186)+(139)+Def. 2.2-1

(189) $CC_2(p_2)$ is defined $\Rightarrow CC_1(p')$ is defined, where $p_2 = F(p')$ (59)

(190) $\Rightarrow (p',\Pi_1'(p')) \in \Pi_1' - \Pi_1$ (131)+(132)

(191) $\Rightarrow \Pi_1'(p') \in N_1$ (51)+Def. 2.2-1

(192) $\Rightarrow I(\Pi_1'(p')) = \Pi_2'(p_2) = n_2$ (91)+(139)+(186)

(193) $\Rightarrow p' = p_1 \Rightarrow CC_1(p_1)$ is defined $\Rightarrow p_1 \in CP$ (110)+(186)+(139)+Def. 2.2-1

(194) p_2 is not the value of an input entry to a structure operation

execution in α_2 and $CC_2(p_2)$ is not defined (186)+(188)+(189)+(193)

(195) For i=1,2, there is no Assign, Update, or Delete execution A such

that $Ent_{\alpha_i}(A,1)$ has value p_i, and $CC_i(p_i)$ is not defined

(186)+(194)+(139)

(196) No Assign, Update, or Delete firing in Ω_i has p_i as a number-1

pointer input (195)+Alg. 4.3-1

(197) $(p_1,n_1) \in \Pi_1' - \Pi_1 \Rightarrow \exists C: NAR_1(C) = (p_1,n_1) \Rightarrow CC_1(p_1)$ is defined

$$(52)+(53)+\text{Def. } 5.2-5$$

(198) $(p_1,n_1) \in \Pi_1$ $\qquad\qquad\qquad$ (197)+(195)

(199) $SM_1'(n_1) = SM_1(n_1)$ $\qquad\qquad$ (198)+(196)+Defs. 3.3-9+2.2-5

(200) Let p be any pointer which appears as the value of an entry in ω_1, and let r be any pointer in dom Π_1' which is not in CP such that $\Pi_1'(r)$ is reachable in U_1' from a node m which is either $\Pi_1'(p)$ or a successor in U_1 of $\Pi_1'(p)$. Prove that $SM_2'(I(\Pi_1'(r))) = I(SM_1'(\Pi_1'(r)))$

(201) There is a path from m to $\Pi_1'(r)$ in U_1', i.e., a sequence of nodes $n_1, n_2, \ldots, n_k$, with $m = n_1$, $\Pi_1'(r) = n_k$, and for $i=1,\ldots,k-1$, n_{i+1} is a successor of n_i; that is, there is an ordered pair (s,n_{i+1}) is $SM_1'(n_i)$ $\qquad\qquad$ (200)+Def. 2.2-2

(202) Let j be such that, for $i=j,\ldots,k$, $\Pi_1'^{-1}(n_i)$ is not in CP. Then for $i=j,\ldots,k$, $SM_1'(n_i) = SM_1(n_i)$ $\qquad\qquad$ (186)+(199)

(203) There is a path from n_j to n_k in U_1 $\qquad$ (51)+(201)+(202)+Def. 2.2-2

(204) $F(p)$ is defined and $(p,\beta_1)\rho(F(p),\beta_2)$ $\qquad$ (200)+(16)+(87)+(88)

(205) $j = 1 \wedge p \notin CP \Rightarrow$ there is a path from m to $\Pi_1'(r)$ in $U_1 \Rightarrow$ there is a path from $\Pi_1'(p)$ to $\Pi_1'(r)$ in U_1 $\qquad$ (202)+(203)+(201)+(200)

(206) $\wedge$ $p \in$ dom Π_1 and $F(p) \in$ dom Π_2 $\qquad\qquad$ (186)+(198)

(207) $\Rightarrow$ there is an arc of P which holds a pointer q_1' in S_1' such that $\Pi_1(p)$ equals or is reachable from $\Pi_1(q_1')$ in U_1

$$(89)+(90)+(51)+(32)+(35)+(36)+(204)+\text{Thm. } 5.3-2$$

(208) $\Rightarrow m = n_j$ equals or is reachable from $\Pi_1(q_1')$ in U_1

$$(201)+(200)+\text{Def. } 2.2-2$$

(209) $j > 1 \vee p \in CP \Rightarrow$ letting p_j and p_{j-1} be such that $n_j = \Pi_1'(p_j)$ and, if $j = 1$, $p_{j-1} = p$, else $n_{j-1} = \Pi_1'(p_{j-1})$, p_{j-1} is in CP, so either

there is an Update execution U such that $V(Ent_{\alpha_1}(U,3)) = p_j$, or

there are two pointers $q_1 \in dom\ \Pi_1$ and $q_2 \in dom\ \Pi_2$ such that (s,n_j) is

in $SM_1(\Pi_1(q_1))$ and $(q_1,\beta_1)\rho(q_2,\beta_2)$

$$(200)+(201)+(175)+(177)+(178)+(202)+(140)+(153)$$

(210) $\Rightarrow$ since $p_j \notin CP$, it is not the value of $Ent_{\alpha_1}(U,3)$, so there are two

pointers $q_1 \in dom\ \Pi_1$ and $q_2 \in dom\ \Pi_2$ such that $(s,n_j) \in SM_1(\Pi_1(q_1))$ and

$(q_1,\beta_1)\rho(q_2,\beta_2)$ $\qquad\qquad\qquad\qquad (202)+(139)$

(211) $\Rightarrow$ there is a q_1' on an arc in S_1' such that $\Pi_1(q_1)$ equals or is

reachable from $\Pi_1(q_1')$ in U_1 $(89)+(90)+(51)+(32)+(35)+(36)+$Thm. 5.3-2

(212) $\Rightarrow n_j$ is a successor of $\Pi_1(q_1)$ in U_1, so n_j is reachable from

$\Pi_1(q_1')$ in U_1 $\qquad\qquad\qquad\qquad (210)+$Def. 2.2-2

(213) There is a pointer q_1' on an arc b in S_1' such that n_j is reachable

from $\Pi_1(q_1')$ in U_1 $\qquad\qquad\qquad (205)+(208)+(209)+(212)$

(214) n_k is reachable from $\Pi_1(q_1')$ in U_1 $\qquad (213)+(203)+$Def. 2.2-2

(215) $n_k = \Pi_1'(r) = \Pi_1(r)$, so n_k is in N_1 $\qquad (201)+(200)+(186)+(198)$

(216) For q_2' the pointer on b in S_2', $U_2.\Pi_2(q_2') \overset{I_1}{=} U_1.\Pi_1(q_1')$ (90)+Def. 2.4-2

(217) $SM_2(I_1(\Pi_1(r))) = I_1(SM_1(\Pi_1(r)))$ $\qquad (214)+(215)+(216)+$Def. 2.4-1

(218) $SM_2'(I(\Pi_1'(r))) = I(SM_1'(\Pi_1'(r)))$ $\qquad (200)+(186)+(199)+(215)+(91)$

(219) Let p be any pointer such that there is a token of value (p,R) or

(p,W) on an arc b in $S_1 \cdot \Omega_1$. Then p appears as the value of an

entry in ω_1, and a token of value $(F(p),R)$ or $(F(p),W)$ is on b

in $S_2 \cdot \Omega_2$ $\qquad\qquad\qquad\qquad\qquad (16)+(19)$

(220) $\Pi_2'(F(p)) = I(\Pi_1'(p))$ $\qquad\qquad\qquad (219)+(92)+(95)$

(221) Let n be any node which equals or is reachable from $\Pi_1'(p)$ in U_1'.

Let q be such that $\Pi_1'(q) = n$. Then $q \in CP \Rightarrow SM_2'(I(n)) = I(SM_1'(n))$

(222) $q \notin CP \Rightarrow SM_2'(I(n)) = I(SM_1'(n))$ (219)+(200)+(218)

(223) $SM_2'(I(n)) = I(SM_1'(n))$ (221)+(222)

(224) $U_2' \cdot \Pi_2'(F(p)) \overset{I}{=} U_1' \cdot \Pi_1'(p)$ (220)+(221)+(223)+Def. 2.4-1

(225) There is a one-to-one mapping I under which for each arc b in P,

Match$((b, S_1 \cdot \Omega_1), I, (b, S_2 \cdot \Omega_2))$

       (9)+(11)+(13)+(15)+(31)+(110)+(219)+(224)+Def. 3.4-1

Finally, it is necessary to prove that, letting the pool components of

$S_1 \cdot \Omega_1$ and $S_2 \cdot \Omega_2$ be Q_1 and Q_2, for every label S of a Select operator,

$\exists p_1: S \in Q_1(p_1) \Leftrightarrow \exists p_2: S \in Q_2(p_2) \Rightarrow U_2' \cdot \Pi_2'(p_2) \overset{I}{=} U_1' \cdot \Pi_1'(p_1)$

(226) There are the same number k of firings of S in Ω_1 and Ω_2 (24)+(25)

(227) Ex(S,k) is initiated in both ω_1 and ω_2 (226)+(24)

(228) For i=1,2, let $\theta_i \varphi_i$ be the prefix of Ω_i in which φ_i is the last
(k^{th}) firing of S. Then S is enabled in $S_i \cdot \theta_i$ Def. 2.3-1

(229) In $S_i \cdot \theta_i$, S is in no pool, and there are no tokens on its output

arcs   (228)+Defs. 3.3-6+2.1-4

(230) $\exists p_i: S \in Q_i(p_i)$ in $S_i \cdot \Omega_i \Leftrightarrow$ for all prefixes Ξ_i of Ω_i longer than θ_i,
$S \in Q_i(p_i)$ in $S_i \cdot \Xi_i$ (228)+Def. 3.3-9

(231) $\Leftrightarrow$ there is no prefix $\Delta_i \varphi_i'$ of Ω_i containing exactly k firings of S
such that tokens appear on the number-1 output arcs of the actor
labelled S in the transition from $S_i \cdot \Delta_i$ to $S_i \cdot \Delta_i \varphi_i'$ Def. 3.3-9

(232) $\Leftrightarrow$ there is no entry in ω_i whose transfer has source Src(Ex(S,k),1)

       (19)+Lemma 4.3-1

(233) $\Leftrightarrow$ there is no token on a number-1 output arc of S in $S_i \cdot \Omega_i$ and
there is no prefix $\Xi_i \varphi_i''$ longer than $\theta_i \varphi_i$ in which φ_i'' removes a

token from such an arc $\qquad$ (228)+(226)+Alg. 4.3-1

(234) $\Rightarrow$ there is no prefix $\Delta_i \varphi_i'$ of Ω_i containing exactly k firings of S

such that tokens appear on the number-1 output arcs of S in the

transition from $S_i \cdot \Delta_i$ to $S_i \cdot \Delta_i \varphi_i'$ (229)

(235) $\exists p_i$: $S \in Q_i(\dot{p}_i)$ in $S_i \cdot \Omega_i$ $\Leftrightarrow$ there is no entry in ω_i whose transfer has

source Src(Ex(S,k),1) (230)+(232)+(234)+(231)

(236) There is no entry in ω_1 whose transfer has source Src(Ex(S,k),1)

iff there is no such entry in ω_2 (6)

(237) $\exists p_1$: $S \in Q_1(p_1)$ in $S_1 \cdot \Omega_1$ iff $\exists p_2$: $S \in Q_2(p_2)$ in $S_2 \cdot \Omega_2$ (235)+(236)

(238) For i=1,2, $S \in Q_i(p_i)$ in $S_i \cdot \Omega_i$ $\Rightarrow$ $S \in Q_i(p_i)$ in $S_i \cdot \theta_i \varphi_i$ (230)

(239) $\Rightarrow$ $S \in Q_i(p_i)$ in Fire($S_i \cdot \theta_i$,S) Def. 3.3-9

(240) $\Rightarrow$ letting $S_i \cdot \theta_i$ be (Γ_i, U_i'', Q_i''), there are tokens of value p_i on

S's number-1 output arcs in $\text{Standard}_\Gamma((\text{Strip}(\Gamma_i,S),U_i''),S)$ Def. 3.3-9

(241) $\Rightarrow$ letting U_i'' be (N_i'', Π_i'', SM_i''), the pair $(s_i, \Pi_i''(p_i))$ is in $SM_i''(\Pi_i''(p_i'))$

where p_i' and s_i are the values of the tokens on S's pointer and

selector input arcs, respectively, in Strip(Γ_i,S) Defs. 3.3-7+2.2-5

(242) $\Rightarrow$ $(s_i, \Pi_i''(p_i)) \in SM_i''(\Pi_i''(p_i'))$ where p_i' and s_i are the values of the

tokens on S's pointer and selector input arcs in Γ_i, which are

removed by φ_i (228)+Def. 3.3-8

(243) $\Rightarrow$ $(s_i, \Pi_i''(p_i)) \in SM_i''(\Pi_i''(p_i'))$ where $p_i' = V(\text{Ent}_{\omega_i}(\text{Ex}(S,k),1))$ and

$s_i = V(\text{Ent}_{\omega_i}(\text{Ex}(S,k),2))$ $\Rightarrow$ $s_1 = s_2$ (12)+Alg. 4.3-1

(244) θ_i is a firing sequence starting in S_i' and $S_i' \cdot \theta_i \mu S_i \cdot \theta_i$

(32)+Thm. 7.1-1

(245) U_i'' is the heap in $S_i' \cdot \theta_i$ (240)+(244)+Def. 7.1-1

(246) Let γ_i be $\eta(S_i', \theta_i)$ and let NAR_i' be the node activation record

derived from θ_i and γ_i. Then U_i'' is the heap determined by γ_i from

U_1 and NAR_1' $\qquad$ (32)+(244)+(245)+Thm. 5.2-1

(247) Let δ_1 be $\eta(S_1', \theta_1\varphi_1)$. Then δ_1 is a causal prefix of ω_1

$\qquad\qquad\qquad$ (228)+Lemma 4.3-2+Alg. 4.3-1

(248) $S \in Q_1(p_1)$ in $S_1 \cdot \Omega_1 \Rightarrow f_1 = Ent_{\delta_1}(Ex(S,k),1)$ is in δ_1 but not in γ_1

and has value p_1' $\qquad\qquad$ (238)+(243)+(228)+Alg. 4.3-1

(249) $\Rightarrow$ for any Update execution U, $[(D(U)$ extends to the end of $H_{p_1'}^{\gamma_1}$ and

$V(Ent_{\gamma_1}(U,2)) = s_1$ iff $f_1 \in D(U)$ in δ_1 and $V(Ent_{\delta_1}(U,2)) = s_1$

$\qquad\qquad\qquad$ (228)+Lemma 5.2-7

(250) iff $Ex(S,k) \in R(U)$ in δ_1 $\qquad\qquad$ (243)+Def. 5.1-8

(251) iff $Ex(S,k) \in R(U)$ in $\omega_1]$ (247)+(36)+(32)+(34)+Lemma 5.2-3+Lemma 5.2-6

(252) $\Rightarrow D(U)$ extends to the end of $H_{p_1'}^{\gamma_1}$ and $V(Ent_{\gamma_1}(U,2)) = s_1$ iff

$Ex(S,k) \in R(U)$ in ω_1 and $V(Ent_{\omega_1}(U,2)) = s_1$ iff $Ex(S,k) \in R(U)$ in ω_2

and $V(Ent_{\omega_2}(U,2)) = s_2$ iff $D(U)$ extends to the end of $H_{p_2'}^{\gamma_2}$ and

$V(Ent_{\gamma_2}(U,2)) = s_2$ $\qquad\qquad$ (247)+(12)+(243)+Def. 6.1-1

(253) $\Rightarrow [D(U)$ extends to the end of $H_{p_1'}^{\gamma_1}$ and $V(Ent_{\gamma_1}(U,2)) = s_1 \Rightarrow$

$p_1 = V(Ent_{\gamma_1}(U,3))$ $\qquad\qquad$ (243)+(246)+Def. 5.2-7

(254) $\Rightarrow p_1$ is the value of an entry in ω_1 and $p_2 = F(p_1)$ $\qquad$ (12)

(255) $\Rightarrow U_2' \cdot \Pi_2'(p_2) \overset{I}{=} U_1' \cdot \Pi_1'(p_1)]$ $\qquad\qquad$ (219)-(224)

(256) $\wedge [\exists U: D(U)$ extends to the end of $H_{p_1'}^{\gamma_1}$ and $V(Ent_{\gamma_1}(U,2)) = s_1 \Rightarrow$

$\exists U: D(U)$ extends to the end of $H_{p_2'}^{\gamma_2}$ and $V(Ent_{\gamma_2}(U,2)) = s_2 \Rightarrow$

letting n_1 be such that $(p_1', n_1) \in \Pi_1'$, and defining (q_1, m_1) by

if $(p_1', n_1) \in \Pi_1$, then $(q_1, m_1) = (p_1', n_1)$,

otherwise, (q_1, m_1) is the unique pair in Π_1 such that

$V(Ent_{\gamma_1}(CC_1(p_1'),1))$ is dynamically descended from q_1 in γ_1

$(s_1, \Pi_1'(p_1)) \in M_1(m_1)$ $\qquad\qquad$ (243)+(246)+Def. 5.2-7

(257) $\Rightarrow$ by the same reasoning as (140)-(154), since $p_1' \in CP$,

$SM_2(m_2) = I_1(SM_1(m_1))$ (243)+(37)+(38)+(39)+(42)+(139)

(258) $\Rightarrow \Pi_2'(p_2) = I(\Pi_1'(p_1))$ (91)+Def. 2.4-1

(259) $\wedge$ q_1 appears as the value of an entry in ω_1 and $\Pi_1'(p_1)$ is a

      successor of $\Pi_1'(q_1)$ in U_1 (243)+(256)+Defs. 5.1-9+2.2-2

(260) $\Rightarrow$ for any node n reachable from $\Pi_1'(p_1)$ in U_1', letting r be such

      that $\Pi_1'(r) = n$, $r \in CP \Rightarrow SM_2'(I(n)) = I(SM_1'(n))$ (185)

(261) $\wedge$ $r \notin CP \Rightarrow SM_2'(I(n)) = I(SM_1'(n))]$ (200)+(218)

(262) $S \in Q_1(p_1)$ in $S_1 \cdot \Omega_1 \Rightarrow U_2' . \Pi_2'(p_2) \stackrel{I}{=} U_1' . \Pi_1'(p_1)$

          (248)+(253)+(255)+(256)+(258)+(260)+(261)+Def. 2.4-1

(263) $S_2 \cdot \Omega_2$ equals $S_1 \cdot \Omega_1$ (225)+(237)+(262)+Def. 7.1-2

(264) P is functional   (1)+(263)

          Q.E.D.

Chapter 8

Summary and Conclusions

This final chapter consists of four sections. The first recapitulates
the goals of the thesis, summarizes the steps taken toward achieving them,
and exhibits the ultimate results. Section 8.2 evaluates these results to
see how well the goals have been met. Section 8.3 presents suggestions
for further research (including several already undertaken by the author).
Section 8.4 completes the thesis with a brief set of conclusions about
the significance of the work which it reports.

8.1 Summary

The primary goal of the thesis is to develop a language L_D and an
interpreter for it, together with a translation algorithm which takes any
well-behaved L_{BV} program P into an L_D program which is equivalent to P
and maximally concurrent. The secondary goal is to render the results in
as general a form as possible, so that they may more easily be applied to
models of concurrent computation other than data flow.

L_{BS}, the data-flow language with structures as storage, offers the
prospect of maximal concurrency, but runs afoul of the problem of non-
functionality: Every L_{BV} program P is functional, so any program equiva-
lent to P must also be functional (precise definitions of functionality
and equivalence are given in Section 2.4). A simple translation of P
into L_{BS} yields a program P' which, on the standard interpreter, may have
more concurrency than P, but also may be non-functional. The solution

pursued here is to retain the structure-as-storage operations, but modify the interpreter so that at least a subset L_D of L_{BS} (which includes P') is functional.

It is argued in Section 3.1 that the only practical way to guarantee functionality is to guarantee freedom from conflict. A program P is conflict-free iff the following is true for every initial state S for P and any two structure operators d_1 and d_2 in P: If there is a firing sequence Ω starting in S in which the ith firing of d_1 potentially interferes with the jth firing of d_2 (Table 3.1-1), which it follows, then those firings are sequenced by S; i.e., the ith firing of d_1 follows the jth firing of d_2 in every firing sequence starting in S. Freedom from conflict actually implies a much stronger condition: determinacy. A program is determinate if all halted firing sequences starting in equal initial states not only produce equal final states, but do so "in the same way". The first major contribution of the thesis is a "scheme", the combination of a restriction on L_{BS} programs and a modification of the interpreter, which eliminates conflict and so guarantees determinacy, hence functionality.

The scheme distinguishes between pairs of potentially-interfering firings in Ω on the basis of whether the two firings are in the same or in different blocking groups (a blocking group is the set of firings in Ω all of which receive their primary pointer inputs from the same program input or Copy or Select firing). In the case of two such firings in the same blocking group, it is assumed that a simple analysis of the program P will reveal whether those firings are sequenced by all initial states of P. It is further assumed that, if necessary, P can be re-written (by inserting sequencers) so that it satisfies the Determinacy Condition, which

is: For any initial state S of P and structure operators d_1 and d_2 in P, if there is a firing sequence starting in S in which the ith firing of d_1 and the jth firing of d_2 potentially interfere and are in the same blocking group, then those firings are sequenced by S. The validity of these assumptions, i.e., the ease of testing and re-writing P, is evaluated later (Section 8.2.1.3). The only immediate concern is for those L_{BS} programs produced by the translation from L_{BV}; it is proven that these do satisfy the Determinacy Condition.

Sequencing a pair of firings in different blocking groups is accomplished by two modifications of the standard interpreter. The first replaces simple pointers with read and write pointers as the values of tokens. The only non-pI-output arcs on which write pointers can appear are the number-1 output arcs of Copy operators. A requirement that every firing of a write-class operator (Assign, Update, or Delete) has a write pointer as its primary input is met by another, simple restriction on programs, the Read-Only Condition. The second, key modification of the interpreter causes it to withhold tokens of read-pointer value (p,R) from the output arcs of a Select operator so long as any arc holds a token of write-pointer value (p,W). The subset of L_{BS} consisting of those programs satisfying both the Determinacy and Read-Only Conditions is the language L_D sought. The standard interpreter with the above two changes is the desired modified interpreter, on which all L_D programs are functional.

Section 3.4 completes the achievement of the primary goal by presenting an algorithm to translate any L_{BV} program P into an L_{BS} program P'. P' is in L_D, and if every L_D program is functional (and P is well-behaved), then P' is equivalent to P (Theorem 3.4-3).

The secondary goal of the thesis is that the proof of L_D's function-
ality should be as general as possible, to make it applicable to other
models of concurrent computation which, like L_{BS}, incorporate the
structure-as-storage operations. To this end, the entry-execution model
is introduced (in Chapter 4). This model focuses just on the definitions
of operations, including how their input-output behaviors may depend on
the order in which they are executed; details of concurrent-control and
local-memory structure are abstracted away. The general form of an entry-
execution model is presented in Section 4.2; Section 4.3 exhibits an
algorithm for constructing the model $EE(L,I)$ from any data-flow language
L and interpreter I. Section 5.1 develops the Structure-as-Storage (S-S)
entry-execution model, to illustrate the technique of defining a set of
interacting operations by constraints on computations; the proof that the
model of L_{BS} on the standard interpreter is an S-S model verifies that
the operations defined are the structure operations in L_{BS}.

Chapter 6 first defines determinacy in entry-execution terms, specif-
ically, a determinate expansion. It then presents a set of axioms on an
expansion which are sufficient to guarantee its determinacy (Theorem 6.4-1).
Chapter 7 proves that the model of L_D on the modified interpreter,
$EE(L_D,M)$, is an S-S model (Theorem 7.1-4) in which every expansion
satisfies the axioms (Lemma 7.2-1, Theorems 7.2-1, 7.2-3, and 7.2-5).
It also proves that if the expansion of program P from $EE(L_D,M)$ is deter-
minate, then P is functional (Theorem 7.3-1). These lead to the following
ultimate results:

Theorem 8.1-1 Every L_D program running on the modified interpreter is functional.

Proof: Theorem 7.1-4, Lemma 7.2-1, Theorems 7.2-1, 7.2-3, 7.2-5, 6.4-1, and 7.3-1.

$\triangle$

Corollary 8.1-1 For any well-behaved L_{BV} program P, the program produced from P by Algorithm 3.4-1 is equivalent to P.

Proof: Theorems 8.1-1 and 3.4-3.

Q.E.D.

This certifies that the first part of the primary goal has been met. The next section evaluates (among other things) progress toward both the other part of this goal (maximal concurrency) and the secondary goal of general applicability of the results.

8.2 Evaluation

This section evaluates the two major contributions of the thesis: (1) the scheme for guaranteeing determinacy and (2) the entry-execution model. The major issue regarding the scheme is whether or not it yields the greatest possible concurrency possible in each program. The simple answer is: No; a more meaningful reply is developed below in the form of a crude cost-benefit analysis, which takes into account implementation considerations. The evaluation of the model is necessarily subjective. The true test of its worth would be a measure of how much the results expressed in its terms can simplify proofs about other concurrent-computation systems with data structures; as yet, no such proofs have been undertaken.

8.2.1 The Scheme

Each of the two parts of the scheme for guaranteeing determinacy incurs a separate cost: Any physical implementation of the standard data-flow interpreter (Section 8.2.1.1 below) must have additional hardware to withhold Select outputs (Section 8.2.1.2); then an arbitrary L_{BS} program must be tested to see if it satisfies the Determinacy and Read-Only Conditions (Section 8.2.1.3). The benefit of the scheme is that it provides the maximum concurrency possible without a far more extensive and costly hardware modification (Section 8.2.1.4).

8.2.1.1 A Data-Flow Processor

A physical implementation of a data-flow interpreter is a data-flow processor. Estimating the added hardware required in the modified data-flow processor necessitates first understanding the envisioned form and function of the standard processor, the four major components of which are diagrammed in Figure 8.2-1 [14].

The Data-Flow Control Unit (DFCU) stores the configuration component of the interpreter state and recognizes enabled actors. It consists of a number of homogeneous, autonomous instruction cells, each of which stores all the information about one actor d:

1. a code for the function associated with d,

2. for each input arc of d, a flag, indicating whether there is a token on that arc, and if so, the value of that token, and

3. for each output arc of d, a destination tag, which

    a. identifies that arc as a particular input arc of a particular other actor, and
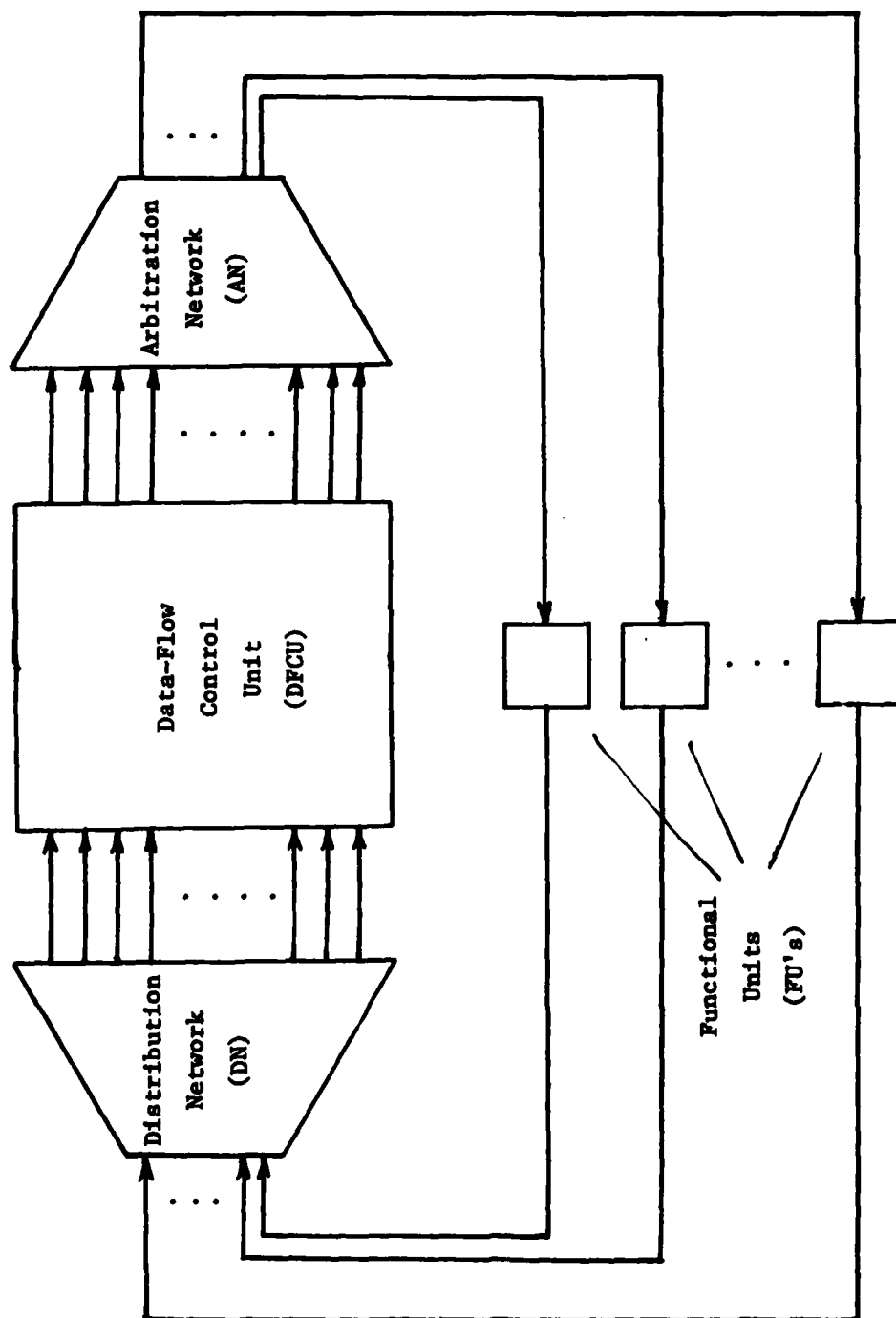
A Data-Flow Processor

Figure 8.2-1

b. indicates whether it is a number-1 or number-2 output arc of d.

Logic within the cell recognizes when d is enabled, i.e., has tokens on the requisite set of input arcs. Whenever this happens, the contents of the cell are bundled into an <u>operation packet</u>, which is transmitted serially through one of several output ports of the DFCU.

All function evaluation is done by the complement of <u>functional units</u> (FU's). There are several distinct types of FU's, and there may be several of each type available on any processor. The different types may include, e.g., an integer arithmetic unit, a floating-point arithmetic unit, and assorted input/output controllers. The number of each type is dictated by economics, with processing power distributed according to demand.

Function codes (as well as data) are meaningless to the DFCU; no internal discrimination is made among actors having different codes. Thus the type of FU needed to evaluate a packet emerging from a DFCU output port is totally unpredictable. One role of the <u>Arbitration Network</u> (AN) is to sort the stream of operation packets at each port into the proper FU types, based on a partial decoding of function codes. Since there may be fewer FU's of a given type than there are DFCU output ports, simultaneous demands for the same FU can arise; arbitration of these demands is the second purpose of the AN.

When an FU receives an operation packet, it executes the indicated function on the ordered set of input values contained in the packet, producing <u>result packets</u>. Each result packet consists of a copy of a result of that execution, paired with one of the destination tags from the received operation packet. Result packets enter the <u>Distribution Network</u> (DN), where they are sorted and directed into the proper one of several

)

input ports of the DFCU. The destination tag in each result packet selects one input arc of one actor; upon entering the DFCU, the result is stored in the proper location of the instruction cell for that actor, where it sets the flag indicating the arrival of a new token.

The Structure Memory (SM) is one type of FU, which executes just the eight structure operations. The issues of how the SM stores a heap and performs operations on it need not be addressed here. One aspect of the SM functioning, however, is essential to explaining the modifications to be made: storage reclamation. Contrary to the simplifying assumption made in the thesis, the sets of nodes and pointers implemented in any physical processor are finite. To postpone saturation of the SM for as long as possible, the storage occupied by the content of a node n will be reclaimed, as will the pointer p to n, whenever n becomes inaccessible. A node becomes inaccessible whenever there is no pointer to it, or to any node from which it is reachable, on an arc in the configuration. Once n becomes inaccessible, p can never again appear on an arc, and so no more operations can ever be performed on n's content. In this case, there is no need to retain that content, so the storage it occupies is reclaimed for use in storing the contents of accessible nodes; similarly, p is made available to point to any new node activated by a subsequent Copy firing.

Of the two principal techniques for detecting inaccessible nodes, the one most easily implemented in hardware is reference counting: For each node n in the SM, there is one reference count (non-negative integer) associated with n and another associated with the pointer p to n. The structure reference count, SRC(n), is the number of nodes of which n is a successor; the execution reference count, ERC(p), is the number of tokens

with value p in the configuration. Whenever $ERC(p) = SRC(n) = 0$, there is no pointer to n on any arc and there is no node from which n is reachable; therefore, n is certainly inaccessible.

At any particular time, there will be only a relatively few non-zero execution reference counts. This argues for storing only these non-zero counts, in what must be an associative memory. This ERC Memory (ERCM) will store associations of pointers with positive integers (keyed on the pointers). The following algorithm correctly maintains the ERCM contents: Whenever an operation packet containing a pointer p leaves the DFCU, find the value associated with p in the ERCM and decrement it by one; if it goes to zero, delete the association. Whenever a result packet with value p leaves the SM, look for an associated $ERC(p)$; if one is found, increment it by one, otherwise add an association pairing p with an ERC of one.

8.2.1.2 The Processor Modifications

Assuming that the above mechanism for maintaining execution reference counts is already available in a standard data-flow processor (as is most likely [1,31]), the necessary modifications are simple: Each pointer value p transmitted outside the SM (through the DN, DFCU, and AN) must be replaced by one of (p,R) or (p,W), either of which is one bit longer than p. The SM must append the correct value of that bit to each result packet generated by the execution of a Copy or Select operator; this value depends on which operation was executed and on whether the destination tag in that packet indicates a number-1 or a number-2 output arc of the operator. Two execution reference counts, $ERC_R(p)$ and $ERC_W(p)$, must be

kept for each pointer p; these are the numbers of tokens with value (p,R) and (p,W), respectively, on arcs of the configuration. The node n to which p points is inaccessible only when $ERC_R(p) = ERC_W(p) = SRC(n) = 0$.

Implementation of the withholding of Select outputs follows the formal specification quite closely: Before result packets with value (p,R) generated by a Select execution leave the SM, look for a non-zero value of $ERC_W(p)$. If none is found, release those packets into the DN. If $ERC_W(p)$ is greater than zero, divert those packets into a separate associative memory, the Pool Memory (PM). Whenever $ERC_W(p)$ is decremented to zero (as the last operation packet with value (p,W) leaves the DFCU), find all result packets with value (p,R) in the PM and release them into the DN.

The formal specification of the modified interpreter (Section 3.3.1) utilizes a two-step state transition, which was claimed to most accurately model the simplest implementation. The basis for this can now be seen: As noted, the only difference between a two-step and a one-step transition arises in the case of a Select firing which inputs the last token of value (p,W) and outputs tokens of value (p,R). In the above implementation, $ERC_W(p)$ will be reduced to zero as the SM starts executing the Select operation, before the result values are known. By the time result (p,R) is produced, $ERC_W(p)$ will be zero, so the result packets will not be withheld. This is just the behavior implied by the two-step transition.

Current trends in technology suggest that the cost of hardware to implement the logic to withhold Select outputs will be far exceeded by that of the additional memory required. Under this assumption, the cost of modifying a standard data-flow processor in accordance with the scheme for guaranteeing determinacy is composed of the following items:

1. The size of the ERCM must be increased to accomodate separate counts for read and write pointers. For each pointer p with any non-zero reference count, both p and the count must be stored in this associative memory. Since the length (in bits) of p is probably greater than that of a reference count, it is most efficient to store just one association, of p with the pair $(ERC_R(p), ERC_W(p))$. Then, e.g., for an SM capacity of 16 million nodes and reference counts of less than 256, the size of the ERCM must be increased by 25% (from 32 to 40 bits per association).

2. An associative Pool Memory must be added. This must store all result packets which are being withheld at any time; its size is impossible to forecast without simulation studies.

3. Data paths through the DN, DFCU, and AN may have to be made one bit wider, but only if their width equals the length of a pointer. In that case, the decision probably would be instead to make pointers one bit shorter, cutting the maximum SM capacity to half as many nodes.

It is very significant that, except for a possible widening of data paths, none of the modifications affects the DFCU, AN, or DN, or the actual structure storage mechanism within the SM; they are restricted primarily to the interface between the SM and the DN.

8.2.1.3 The Program Restrictions

Modifying the interpreter is only one of two parts of the scheme for guaranteeing determinacy of a program P. The other is the requirement that P is in L_D, i.e., satisfies the Determinacy and Read-Only Conditions.

Within the narrow scope of the goal of the thesis, this part of the scheme has zero cost: it has been proven that Algorithm 3.4-1 translates every L_{BV} program into an L_D program. For an arbitrary L_{BS} program P, however, establishing whether determinacy is guaranteed requires examining the program. The growth of the computational effort of this examination with program size is an important "figure of merit" for the scheme.

The essence of the Read-Only Condition is to force the primary input to every write-class execution to be a write pointer. Compliance with this restriction is so trivially checked in hardware that any effort spent analyzing the program for it would be extravagant. It is assumed therefore that the arrival at the SM of any operation packet containing the code for a write-class operation along with a read pointer as the number-1 input causes an exception, indicating that the program is not guaranteed determinate.

The Determinacy Condition concerns every two potentially-interfering firings in a common blocking group in any firing sequence starting in any initial state S of P. By the Static/Dynamic Group Relationship, a firing of actor d_1 and a firing of actor d_2 are in the same blocking group in any firing sequence only if d_1 and d_2 are in the same maximal pointer distribution group (m.p.d.g.) in P (Definition 3.2-1). Since one of d_1 and d_2 must be write-class, the Read-Only Condition implies that that m.p.d.g. must be $G(K(C,1))$ for some Copy operator C. Table 3.1-1 may show that, because of their operations, no firings of d_1 and d_2 can potentially interfere. If d_1 is an Update or Delete and d_2 is a Select, Update, or Delete, firings of them potentially interfere only if they have the same selector input; it may be possible to prove that this never occurs for

firings in the same blocking group. Otherwise, all firings of d_1 and d_2 in the same blocking group must be sequenced by all initial states of P. As mentioned at the end of Section 3.3, the following is believed to be sufficient for such sequencing, if P is well-formed: either d_1 and d_2 are in separate branches of the same conditional construct, or there is a directed path in P from one to the other.

Therefore, for every Copy operator C in P, every write-class operator in $G(K(C,1))$ must be checked against every other structure operator in $G(K(C,1)) \cup G(K(C,2))$, first to see if there can be potentially-interfering firings of those actors in a common blocking group, and then if so, to see if those firings are sequenced. Thus the effort expended for each m.p.d.g. may grow as fast as the square of the number of structure operators in it. It is reasonable to expect, however, that this number would be bounded from above by some relatively small constant. If so, then the total effort required to determine whether an arbitrary L_{BS} program is guaranteed determinate is proportional to the number of Copy operators in it; i.e., grows linearly with program size.

This completes the projection of the costs of the scheme for guaranteeing determinacy. Next it is argued that the benefit of the scheme is that it provides the maximum concurrency possible without far more extensive modifications of the DFCU.

8.2.1.4 Degree of Concurrency

Section 2.3.3 (q.v.) provides a measure of concurrency and uses it to compare the L_{BV} program AlterV2 and a similar, but non-functional L_{BS} program AlterS2. That analysis, based on Assumptions 2.3-1 and 2.3-2,

concludes that the minimum elapsed time required to "correctly" execute

AlterS2 is 2S less than that for AlterV2, which is at most 8S (S is the

time required to execute a Copy); this is a reduction of at least 25%.

Translating AlterV2 into L_D produces AlterS2' (Figure 3.4-2), which, when

run on the modified interpreter, is functional, and so always executes

"correctly". The only difference between AlterS2 and AlterS2' is the

insertion in the latter of a sequencer, which forces Select S_3 to fire

after Update U_1. That sequencer is on one of the maximal-execution-time

paths. Therefore, the minimum elapsed time required to execute AlterS2'

is greater than that for AlterS2 by the execution time of a sequencer.

A sequencer should take less time to execute than a Copy, probably much

less time. Hence the improvement in elapsed time from AlterV2 to AlterS2'

will probably be a few per cent less than the improvement from AlterV2

to AlterS2, but it should still be at least 20%.

L_D programs on the modified interpreter are in general not maximally

concurrent. Loss of concurrency occurs whenever the firing of one actor

is delayed, even though all its inputs are available, until another actor

has fired, but the two firings do not potentially interfere. The circum-

stances under which such losses occur, and the further processor modifi-

cations necessary to reduce their frequency, are discussed below, in the

two cases that the firings in question are in the same or in different

blocking groups.

With an optimal algorithm for achieving the Determinacy Condition,

concurrency is lost between firings in the same blocking group of two

actors d_1 and d_2 only in the following case: one is an Update or Delete,

the other is a Select, Update, or Delete, and it could not be proven that

every two firings of d_1 and d_2 in any common blocking group would have different selector inputs. This implies that the unnecessary delay of a firing of d_1 until after a firing of d_2 can only be detected and corrected as it occurs, i.e., by processor hardware. An unnecessary delay is indicated by the conjunction of the following three circumstances: (1) both inputs to d_1 are available (i.e., are stored in the instruction cell for d_1 in the DFCU), (2) the selector input to d_2 is available (since d_1 is being delayed unnecessarily; i.e., the selector inputs are known to be distinct), and (3) another input to d_2 is not available (otherwise d_2 would have fired). Avoiding the delay requires checking that the cells for all Update, Delete, or Select operators in the same m.p.d.g. as d_1 store either a different selector input or a different pointer input than d_1's cell. In general, this calls for a pair of comparators between every two cells in the DFCU which can hold structure operators, a very expensive proposition.

A firing of d_1 in one blocking group can be delayed until after a firing of d_2 in another blocking group without regard to whether the two could potentially interfere, as in the following case: A Select firing has generated a result packet containing value (p,R) which is destined for d_1's input, but that packet is being withheld until $ERC_W(p)$ goes to zero, which will not happen at least until the firing of d_2. Thus even though the input to d_1 has been generated and could have arrived at the cell for d_1, it will not do so until d_2 fires. This happens even if those firings could never potentially interfere, e.g., if d_1 is a Fetch and d_2 is a Delete.

Such inter-blocking-group concurrency losses may be reduced by a further refinement of the scheme: The two classes of structure operations, read and write, are partitioned into five subclasses. The write class is

split into the write-value (Assign) and the write-branch (Update and Delete) subclasses. Since a Copy (unfortunately) reads both value and branches, the read class must be divided into three subclasses: read-value (Fetch), read-branch (Select, First, and Next), and read-all (Copy). There are five corresponding types of tagged pointers. A Copy operator must have three distinct groups of output arcs (write-value, write-branch, and read), and a Select must have four (the three read subclasses plus a control output). Finally, there must be three execution reference counts: one for write-value pointers, one for write-branch pointers, and one for all read pointers.

The major costs of these further modifications to each of the components of the processor are as follows: DFCU, AN, and DN – a pointer data path which is two bits wider (or a reduction in SM capacity of 75%), and destination tags which are one bit longer (to distinguish four groups of output arcs instead of two). SM – one more ERC to store (a size increase of 20%, assuming as before 24-bit pointers and 8-bit ERC's).

In conclusion, the following claims are made about the degree of concurrency among structure operato· under certain "ground rules":

Only the SM can be modified – The ori   al modified processor is maximally concurrent (although the maximum SM capacity may have to be reduced to half as many nodes, if the standard processor data paths are not wider than a pointer).

The DFCU may be modified slightly – The refinement sketched above, requiring one extra bit in each destination tag, is maximally concurrent.

Under at least the first ground rule, the thesis meets its primary goal.

8.2.2 The Model

The secondary goal of the thesis is to make the proofs of the correct-
ness of the scheme for guaranteeing determinacy as general as possible,
in hopes of shortening future proofs about concurrent-computation systems
other than data flow which use the scheme. The basic proofs concern how
the outputs of an execution depend on the inputs to it and to preceding
executions; any medium for their expression should, therefore, convey this
information with as few extraneous details as possible.

The entry-execution model is a good such medium. In each computation,
there is one entry for each value input to every execution, each value
output by every execution is listed in at least one entry, and execution
order is indicated by initiation order. The steps in using this model to
prove that determinacy is guaranteed in any system are listed at the start
of Chapter 4. As evidenced by the length of Chapter 7, this likely will
not be an easy chore; only experience will tell if it is easier than it
would be without the results about entry-execution models developed in
Chapter 6.

An important additional application of the entry-execution model is
in describing the behavior of physical processors. A computation can be
viewed as a behavior of a data-flow processor, under the following inter-
pretation: An execution corresponds to an operation packet. An operation
packet is "grown" in the instruction cell for an actor as an accumulation
of result packets. When a full set of result packets has been received,
the actor is fired; i.e., the operation packet is sent to a Functional
Unit. There it generates result packets, which are sent through the DN
to the proper incipient operation packets in the DFCU. Thus the *entry*

of a result packet into the DN marks the transfer of a value from an output of one operation packet to an input of another. Substituting "execution" for "operation packet", this is just the definition of an entry. Under this interpretation, the algorithm for reconstructing the firing sequence $\Phi(\omega)$ from the entry sequence ω (Definition 4.3-4) reads: Accumulate the entries (result packets) which are the inputs to an execution (operation packet); when the initiating (last) one arrives, register a new firing. This is just the principle of operation of the DFCU.

A computation (sequence of entries) potentially provides a much more precise description of processor behavior than a firing sequence. The latter implies that at most one FU is active at a time: An FU is active for as long as it takes to execute the operation of one firing, and only after that is finished is another FU activated by the next firing in the sequence. In a real processor, however, an FU is active for some of the time between the completion of the operation packet for a firing and the arrival at the DN of the first of the associated result packets. On the above interpretation of computations, this active period corresponds to the interval between the initiation of an execution and that execution's first output entry. Since such intervals can overlap, a computation can describe concurrent activity by several FU's.

Unfortunately, the full descriptive potential inherent in computations is not realized by the model EE(L,I) of data-flow language L and interpreter I, as the following demonstrates: If the data-flow processor implementing I is running program P, then for any actor d in P, there is an instruction cell in the DFCU. Initially, some number n of d's input arcs are empty. That cell will first recognize d as enabled at some time

after the time t_0 at which the nth result packet destined for the cell

enters the DN. The cell will then send an operation packet to an FU, where

result packets are generated; the first of these enters the DN at some time

$t_1 > t_0$. It is always possible that no other result packet enters the DN

between t_0 and t_1 (unless, as explained later, d is a Select and this is

the modified processor).

Each job J in the expansion of P from $EE(L,I)$ consists of all the

computations generated by I when started in any of some equivalence class

of initial states for P. Ideally, those computations would be the

descriptions of all possible behaviors of the processor implementing I

when started in any of those initial states. If this were so, then by the

above, for any $\alpha f\delta g \in J$ in which f is the nth (initiating) entry to $Ex(d,1)$

and g is any output entry of that execution, αfg would be in J (i.e., none

of the entries in δ would have to appear between f and g). But

Definition 4.3-5 imposes an additional restriction on computations in J:

for every input entry h to g's target execution, the execution of which h

is an output entry must be initiated in αf. This is an artificial restric-

tion, not reflecting any physical processor characteristic, and so inval-

idates $EE(L,I)$ as an accurate description of the implementation of I.

A model in which that restriction is removed would provide a descrip-

tion of the standard processor which is both accurate and significantly

more precise than firing sequences. For the modified processor, the

restriction cannot be removed entirely, as there is a case in which certain

result packets may have to enter the DN between t_0 and t_1: If the result

packet entering at t_1 has value (p,R), $ERC_W(p) > 0$ at t_0, meaning that

some instruction cell is storing the value (p,W), and that cell needs more

inputs before it is enabled. The following redefinition contains a
suitable restriction, producing an accurate description of the behavior
of either processor:

Definition 8.2-1 Let S be any initial state for a data-flow program P,
and let Ω be any halted firing sequence starting in S. Then the set
$J_{S,\Omega}$ of computations for $\underline{S}$ and $\underline{\Omega}$ consists of each permutation β of $\eta(S,\Omega)$
which satisfies all of the following:

   1. $\Phi(\beta)$ is the reduction of Ω.

   2. β is causal.

   3. For every prefix αf of β, let θ be the prefix of Ω whose reduction
       is $\Phi(\alpha)$, let the destination in $T(f)$ be $Dst(Ex(d,k),j)$, and let b
       be the arc given by:

           $d \notin DL \Rightarrow$ b is the number-j input arc of the actor labelled d

           $d = (c,n)$ and $c \neq$ "OD" $\Rightarrow$ b is the number-n input arc of the
              actor labelled c

           $d = (c,n)$ and $c =$ "OD" $\Rightarrow$ b is the number-n program output arc

       Then there is a token on b in $S \cdot \theta$.

$\triangle$

   This revision affects the proofs only of the following: Lemma 4.3-3,
Lemma 5.3-2, Lemma 7.2-8, and Theorem 7.2-5. While the effects on the
first three of these are minor, the last one, the proof of persistence,
would be extremely difficult without the original restriction. As it has
been proven that every expansion in the original model is determinate,
however, it should be possible to work backwards to prove that every
expansion in the revised model is persistent. Since all other proofs

apply to the revised model (with minor reworking), every expansion in the revised model is determinate. This is a very powerful statement about the behavior of a correct physical implementation of a data-flow interpreter.

8.3 Suggestions for Further Research

This section presents two types of suggestions: (1) resolving questions previously raised, about the scheme and the model as presently constituted, and (2) exploring proposed extensions.

8.3.1 Open Questions

Several unsolved problems have been noted in the course of the thesis, many of them in the just-concluded section evaluating the proven results. These are summarized below:

1. Confirm a general syntactic test for the Determinacy Condition. It is known how to identify those pairs of actors of which all firings in the same blocking group must be sequenced. It is believed that in a well-formed program, it is sufficient that for every such pair, either each actor is in a separate branch of a conditional construct or there is a directed path between them. This claim has thus far successfully resisted extensive efforts at a proof.

2. Devise a syntactic means of recognizing potential hangups. The following line of attack seems promising: The blocking diagram for program P is a graph with one node for each Copy and Select operator in P. An arc is drawn from the node for Select S to that for Copy C iff there is a directed path in P from S to any actor in the m.p.d.g. $G(K(C,1))$. An arc is drawn from the node for C to the node for S iff a firing of S could ever output the same pointer as a firing of C.

A hangup occurs when (a) output tokens are being withheld from S's output arcs until an actor in $G(K(C,1))$ fires, but (b) that actor cannot fire until tokens appear on S's output arcs. This situation implies the existence of a directed cycle in the blocking diagram.

3. Prove that every expansion in the revised entry-execution model of L_D on the modified interpreter (just described in Section 8.2.2) is persistent, hence determinate.

4. Conduct more detailed studies comparing the benefits and costs of L_D on the modified interpreter against those of L_{BV} on the standard interpreter. The major benefit claimed for L_D is increased concurrency. Classes of "toy" programs in which structure operations predominate may yield analytical comparisons, such as that between AlterV2 and AlterS2; simulation of a set of real programs is needed to discover the actual concurrency advantage, if any. Establishing the incremental cost of the modified processor must await determination of the base cost of the standard processor, including the SM.

5. Resolve the problem introduced by the ability to use Structure-as-Storage operations to build a heap containing directed cycles (directed cycles cannot be constructed using just the Structure-as-Value operations [12]). The presence of cycles does not directly impact the determinacy scheme, but it does confound the reference-counting method of storage reclamation: In a directed cycle in which every node is inaccessible, every node still has a predecessor, hence a non-zero structure reference count; thus the storage for cycles is never reclaimed. Several solutions suggest themselves:

a. Run only programs which are equivalent to L_{BV} programs. Such a program does not build cycles, but is more concurrent than its L_{BV} counterpart.

b. In the same vein, encase all write-class operators in a set of procedures, similar to those defined in [21] to implement a relational data base. It may then be possible to prove that no program using just these procedures can build a cycle.

c. Prevent or mark each cycle dynamically, during execution of the instruction which would create it. This involves finding all nodes reachable from that pointed to by the number-3 input of an Update, to see if the node pointed to by the number-1 input is among them.

d. Enhance the SM with an incremental garbage collector [16]. This is an independent processor, which traces and marks all access-ible structures (i.e., beginning with pointers having non-zero ERC's), and then reclaims all unmarked nodes.

6. Design a Structure Memory, a Functional Unit which directly and efficiently executes a set of structure operations.

8.3.2 Extensions

This section discusses several partially-developed extensions of both the scheme and the model. The major extension of the scheme is motivated by a significant implementation inefficiency inherent in the structure operations presented earlier. The problem is illustrated by Figure 2.3-5d, which is the final state in a sequence starting in the initial state of program AlterS shown in Figure 2.3-4. The heap in the final state

contains both the component which was a program input (nodes m_1 and m_2) and the component which is a program output (nodes n_1 and n_2).

This portends inefficiency in a physical SM if, in the initial state, $SRC(m_1) = 0$ and $ERC_R(p_1)+ERC_W(p_1) = 2$, where p_1 is the pointer to m_1. I.e., there are only two tokens with value (p_1,R) or (p_1,W) in the DFCU, and these are on arcs in AlterS. As a consequence, in the final state, $SRC(m_1) = 0$ and $ERC_R(p_1) = ERC_W(p_1) = 0$, so the storage for m_1 can be reclaimed. Included in that reclamation is an implicit Delete of all branches emanating from m_1; i.e., for each successor n of m_1 (including m_2), the number of branches terminating on n, which is $SRC(n)$, is reduced by one. Assuming that, in the initial state, $SRC(m_2) = 1$ and $ERC_R(p_2)+ERC_W(p_2) = 0$, where p_2 is the pointer to m_2, m_2 will be inaccessible, hence eligible for reclamation, as soon as m_1 is reclaimed. The program output component (n_1 and n_2) is almost identical to the program input component. Therefore, the effect of the program is to copy its input component, with a minor alteration, and then discard that component. It is far more sensible to make the minor change directly to the input component, and then output the program input p_1. This would save the considerable efforts involved both in copying the input component and in reclaiming it.

Altering m_2 (changing its value from 2 to 3) would require an Assign firing which has p_2 as an input. That pointer can only be obtained as the output of a Select firing which has p_1 as an input. But an Assign firing must have a write pointer as input, and a Select outputs only read pointers. This dilemma can be resolved by introducing a ninth structure operation, Modify, differing from Select in that it outputs write pointers.

The Modify operation permits a program to avoid copying a node which will immediately become inaccessible; unfortunately, it also defeats the scheme for guaranteeing determinacy. This can be rectified by a simple extension of the scheme. Reviewing its original explication (Section 3.2), the possibilities for potential interference between firings in different blocking groups are set down as the Potential-Interference Assumption. This presupposes that any write-class firing in firing sequence Ω is in $B_\Omega(Tg(C,n))$ for some Copy operator C, which is no longer valid with the introduction of the Modify operation. Hence, a slightly different Potential-Interference Assumption is needed:

> Given a firing sequence Ω and two distinct blocking groups $B_\Omega(e)$ and $B_\Omega(e')$, some firing in one group potentially interferes with some firing in the other iff:
>
> 1. $e = Tg(d_1, n_1)$ for some n_1, where d_1 is a Copy or Modify operator,
> 2. $e' = Tg(d_2, n_2)$ for some n_2, where d_2 is a Select or Modify, and
> 3. the n_2^{th} firing of d_2 outputs the same pointer as the n_1^{th} firing of d_1.

The strategy adopted is to sequence all firings in one group with respect to all firings in another group if it is assumed that some firing in one group potentially interferes with some firing in the other.

All the firings in blocking group $B_\Omega(e_2)$ are sequenced after all firings in $B_\Omega(e_1)$ by the Group Sequencing Technique, consisting of two rules:

> I. The first tokens with tag e_1 appear before the first tokens with tag e_2.

II. The first tokens with tag e_2 do not appear while there are tokens with tag e_1 in the configuration.

Rule II leads to the Blocking Discipline: For any pointer p, no token whose value is the tagged pointer $TP(p,e_2)$ appears on any arc if there are tokens of value $TP(p,e_1)$ on any arc. At most one Copy firing outputs p, and no other firing can output p before it does; therefore, tokens never need be withheld from output arcs of a Copy. For any tag $e_2 = Tg(S,j)$ where S is a Select operator, by the Potential-Interference Assumption, the firings in $B_\Omega(e_2)$ are to be sequenced after those in $B_\Omega(e_1)$ only if $e_1 = Tg(C,n)$ for Copy or Modify operator C. For tag $e_2 = Tg(M,j)$ where M is a Modify operator, $e_1 = Tg(A,n)$ where A is either a Copy or Modify operator or a Select operator. It is argued in Section 3.3.1 that the pointer-valued tokens output by the nth firing of structure operator d in Ω need have only one of two tags: W if there may be a write-class firing in $B_\Omega(Tg(d,n))$ (i.e., if d is a Copy or Modify), or R otherwise. Therefore, Rule II is enforced as follows:

For any pointer p,

no tokens of value (p,R) are placed on output arcs of a Select operator while there are tokens of value (p,W) in the configuration, and

no tokens of value (p,W) are placed on the output arcs of a Modify operator while there are tokens of value (p,W) or (p,R) in the configuration.

Rule I is enforced by a combination of the two more fundamental rules:

Ia. If, for i=1,2, $e_i = Tg(d_i, n_i)$, the first tokens with tag e_1 appear

before the first tokens with tag e_2 iff the n_1^{th} firing of d_1

precedes the n_2^{th} firing of d_2.

Ib. The n_1^{th} firing of d_1 always precedes the n_2^{th} firing of d_2.

Rule Ia implies that the third component of the modified interpreter state,

Q, should contain, for each pointer p, a first-in, first-out queue of actor

labels, rather than a pool, as the following demonstrates: The mth firing

of Select S and the nth firing of Modify M, in that order, may attempt to

output tokens of value $TP(p, Tg(S,m))$ and $TP(p, Tg(M,n))$ while there are

still tokens of value $TP(p, Tg(C,j))$ for Copy operator C in the configur-

ation. By Rule II, none of the former tokens can appear on output arcs of

S or M until the last token with value $TP(p, Tg(C,j))$ disappears. At that

time, by Rule Ia, the tokens of value $TP(p, Tg(S,m))$ must appear on S's

output arcs first, since S fired before M. Tokens cannot be placed on M's

output arcs at this time, by Rule II again. Thus the necessity of

remembering the order in which actor labels are added to Q(p).

Rule Ib applies to any two firings of actors d_1 and d_2 which output

the same pointer if one of the actors is a Copy or Modify and the other is

a Modify or Select. A Copy firing always precedes any other firing which

outputs the same pointer. If one actor is a Modify and the other is a

Modify or Select, there are two cases to consider: the two firings either

do or do not have identical pointer and selector inputs. If they do,

Rule Ib becomes: Given that the n_1^{th} firing of d_1 and the n_2^{th} firing of

d_2 in some firing sequence have the same pointer and selector inputs, at

least one operator is a Modify, and the other is a Modify or Select, those

firings must be sequenced. Significantly, an equally-true statement is

obtained by replacing "Modify" in the foregoing with "Update or Delete".
The determinacy scheme guarantees the latter sequencing, so the problem
of enforcing Rule Ib in this case is fundamentally no different than one
which has already been solved. That solution is easily adapted to handle
the new Modify operator as follows:

A. Modify (like Update and Delete) is a write-class operator; i.e.,
   requires a write pointer input.

B. The Determinacy Condition is extended to require sequencing of any
   two Select or Modify firings in the same blocking group which may
   have the same selector inputs, if at least one is a Modify firing.

The extension of the determinacy scheme to accomodate the Modify
operation, as developed to this point, is summarized below:

1. Modify operates like Select except that it requires a write-pointer
   input and produces a write-pointer output, and any tokens of value
   (p,W) are withheld from the output arcs of a Modify so long as there
   are tokens of value (p,W) or (p,R) in the configuration.

2. The third state component Q consists of a queue of actor labels for
   each pointer.

3. The Determinacy Condition is extended as just noted.

It is believed that these changes will yield the following guarantee:
For any equal initial states S_1 and S_2 and halted firing sequences Ω_1 and
Ω_2, $S_2 \cdot \Omega_2$ equals $S_1 \cdot \Omega_1$, unless there is a Modify d_1 and a Modify or Select
d_2, and an n_1 and n_2, such that the n_1^{th} firing of d_1 and the n_2^{th} firing
of d_2 are not sequenced and in Ω_1, they have the same pointer output but
different pointer or selector inputs. A suggestion for further research
is to use the techniques and results of the thesis to prove this formally.

If two unsequenced Modify or Select firings can output the same pointer p to node n given pointers to different nodes m_1 and m_2, it is because there are branches to n from each of m_1 and m_2. This condition is easily detected, as the structure reference count SRC(n) will be greater than one. Thus it is possible to determine, at a Modify firing, whether another Modify or Select firing with a different pointer (or selector) input could subsequently output the same pointer. Rather than try to sequence these two firings, with their different inputs, the need for sequencing can be eliminated altogether, by making the Modify operation more sophisticated: Before a Modify firing outputs pointer p to node n, SRC(n) is checked. If it is 0 or 1, the pointer is output as indicated above (i.e., it is withheld until $ERC_R(p)+ERC_W(p)$ is zero). If SRC(n) > 1, the effect of the Modify firing becomes as if it had been replaced with the four actors depicted in Figure 8.3-1a. That is, a copy n' is made of node n and n' is made the 's'-successor of the node m (Figure 8.3-1b); then the pointer to n' is output immediately. The component rooted at m after this alternative Modify action (called _automatic_ _copying_) equals that before the action, and the component rooted at n' equals that rooted at n. The only difference is that every other firing which outputs a pointer to n' must follow this Modify firing which activated n'; thus Rule Ib is obeyed. A formal proof is needed of the claim: with automatic copying by Modify operators, all programs are functional; furthermore, every L_{BV} program can be translated into an equivalent one of these programs, which not only may have more concurrency, but may activate fewer nodes.
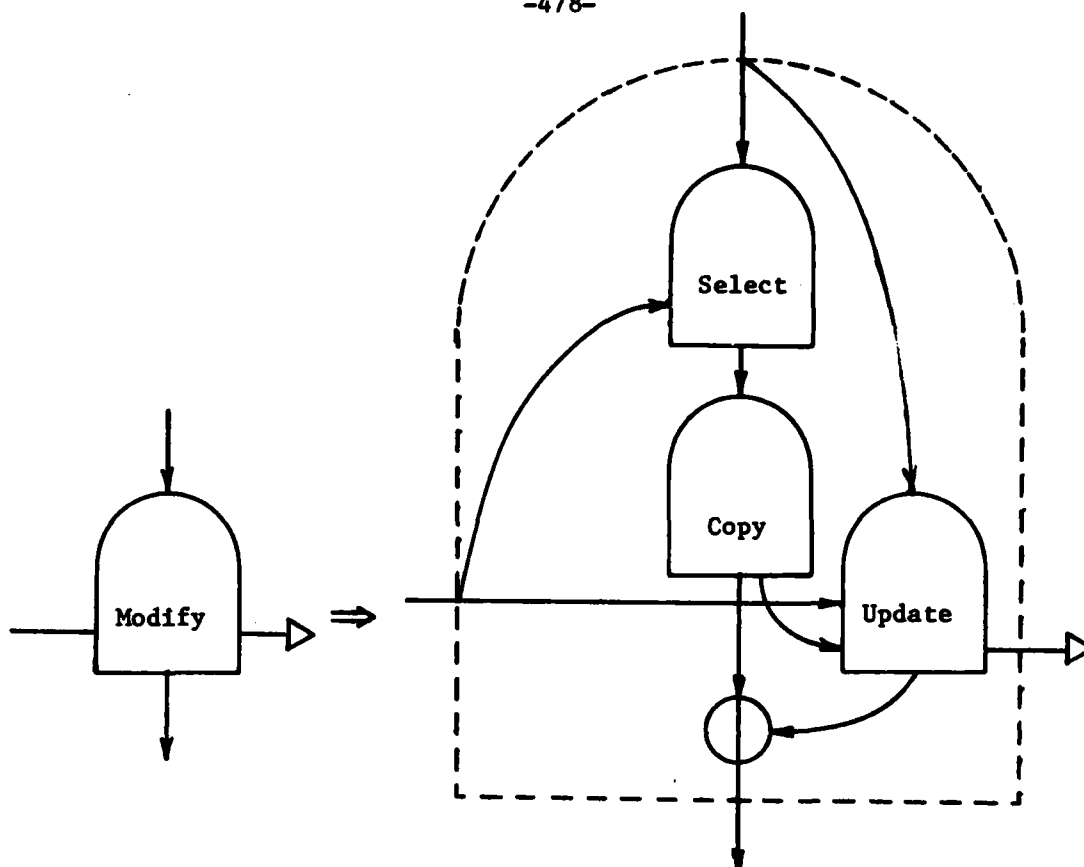
Effective Dynamic Substitution

(a)
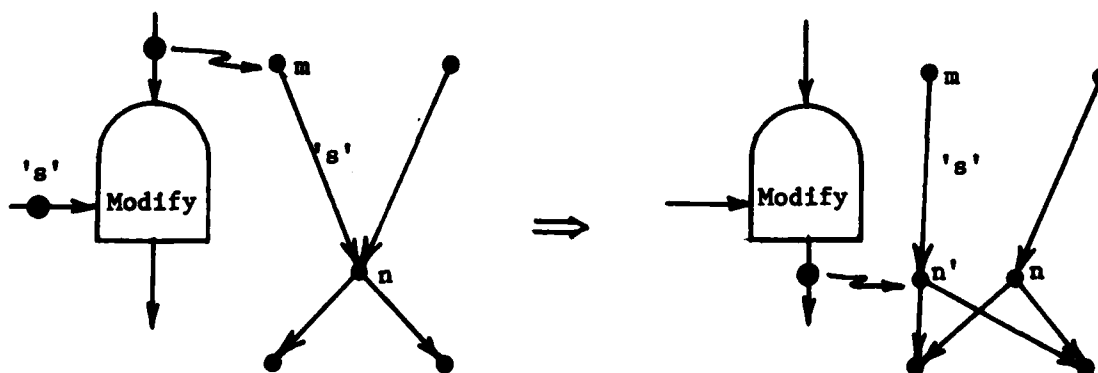
Effect on Heap

(b)

Automatic Copying

Figure 8.3-1

Two final extensions of the determinacy scheme are less well developed. The first is the accomodation of procedures [12]. The major problem introduced is that a single blocking group may contain firings of actors from several different procedures. Any syntactic test for the Determinacy Condition must require only that each individual procedure be verified independently; it would be unworkable if each change to a procedure entailed re-examining every procedure which it may ever call or which may ever call it. The second extension is the use of structure operations to obtain a conveniently-controlled non-determinacy. A most exciting prospect is to allow a node n to have an attribute (the shared attribute), which would disable automatic copying, so that a Modify firing could output a write pointer to n even when $SRC(n) > 1$. From the argument given above, different firing sequences starting in equal initial states give rise to unequal final states only if two firings with different pointer or selector inputs output pointers to the same shared node in a different order.

The only developed suggestion for extending the entry-execution model is a consequence of storage reclamation. Specifically, the finiteness of the set of pointer values in any physical implementation means that one pointer may be the output of several Copy firings in a single firing sequence. A correct implementation never allows a Copy firing to output a pointer to an accessible node; every Copy firing outputs a pointer which either does not point to any node or points to an inaccessible node, whose storage has presumably been reclaimed. Modeling such a correct implementation requires two initial steps:

6 OF 6

END
DATE
FILMED
6-80
DTIC
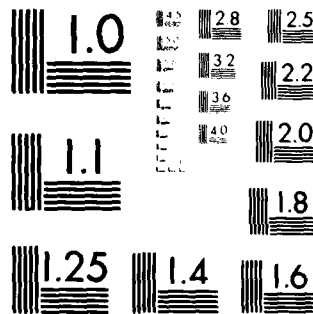
MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS 1963-A

1. Redefine the access history, which is more properly a history of the accesses to a unique node than the history of accesses using a unique pointer. Two entries with value p in a computation ω should be in the same access history iff they are not separated in ω by the initiating entry of a Copy execution with output entries of value p.

2. Add a new constraint to the definition of a Structure-as-Storage model. This should reflect the fact that if the nth firing of Copy or Select operator S in firing sequence Ω outputs p, then there are tokens of value p in all subsequent states until the last firing in blocking group $B_\Omega(S,n)$. Therefore, no Copy firing should output p between the nth firing of S and the last firing in $B_\Omega(S,n)$.

These revisions must then of course be propagated through the proofs in Chapters 5, 6, and 7.

8.4 Conclusions

Those results of the thesis which are felt to be significant original contributions are listed briefly below:

1. A state-oriented, but non-graphical, definition of a complete set of primitive structure operations (of which First and Next are original) (Section 2.2).

2. A definition of equality of data-flow interpreter states with heap components, and the concomitant definitions of functionality and equivalence between two languages (Section 2.4).

3. The language L_D, the modified interpreter, and a translation algorithm to take any well-behaved L_{BV} program into an equivalent L_D

program which, at least under certain ground rules, is maximally concurrent.

4. A new model of concurrent computation, which offers a memoryless (non-state-oriented) representation of the essential order-dependent input-output behavior of structure operations. This expresses what is common among all systems of concurrent computation over data structures, without regard to their idiosyncratic control and local-memory structures (Chapters 4 and 5).

5. A definition of determinacy and a set of axioms which are proven sufficient for determinacy, all using the entry-execution model, and hence all applicable to any concurrent computations over data structures (Chapter 6).

The proposed extensions should prove even more significant. The Modify operator can eliminate most of the need for the Copy operator, by automatically copying a node only when necessary for determinacy. The shared attribute for a node, which defeats automatic copying, should provide for natural solutions to non-determinate problems, such as the airline reservation system [5], in which the integrity of the data base is readily assured.

Appendix A

Proof of Theorem 2.4-1

Theorem 2.4-1 The "Match" relation is symmetric and transitive.

Proof:

Key definitions: Def. 2.2-2 - successor, reachability, path

Def. 2.4-1 - equal components in a heap

Def. 2.4-2 - Match

Prove symmetry first. Let $S_1 = (\Gamma_1, U_1)$ and $S_2 = (\Gamma_2, U_2)$ be any two interpreter states, where $U_1 = (N_1, \Pi_1, SM_1)$ and $U_2 = (N_2, \Pi_2, SM_2)$. Let b_1 and b_2 each be an arc from the programs of which Γ_1 and Γ_2 respectively are configurations. Then for any one-to-one mapping $I: N_1 \rightarrow N_2$, prove that $\text{Match}((b_2, S_2), I, (b_1, S_1)) \rightarrow \text{Match}((b_1, S_1), I^{-1}, (b_2, S_2))$.

(1) Since I is one-to-one, $I^{-1}: N_2 \rightarrow N_1$ is also one-to-one.

(2) Let $m_1 \in N_1$ and $m_2 \in N_2$ be any two nodes for which $U_2.m_2 \overset{I}{=} U_1.m_1$. Then $m_2 = I(m_1)$, and for each node n_1 equal to or reachable from m_1 in U_1, $SM_2(I(n_1)) = I(SM_1(n_1))$.

Now prove the following preliminary result:

A: $\forall n \in N_1$, n is reachable from m_1 in $U_1 \rightarrow I(n)$ is reachable from m_2 in U_2 and $\forall n \in N_2$, n is reachable from $m_2 \rightarrow I^{-1}(n)$ is reachable from m_1

Proof of A is by induction on the length k of the shortest path from m_1 to n, or from m_2 to n.

Basis: $k = 1$.

(3) $SM_1(m_1) = \{v, (s_1, r_1), \ldots, (s_j, r_j)\}$ iff

$SM_2(I(m_1)) = \{v, (s_1, I(r_1)), \ldots, (s_j, I(r_j))\}$ (2)

(4) $\forall n \in N_1$, n is reachable from m_1 by a shortest path of length 1 $\Rightarrow$ n is

a successor of m_1 $\Rightarrow$ $\exists i: r_i = n$ [(3)] $\Rightarrow$ $I(n) = I(r_i)$ is a successor

of m_2 [(2)+(3)] $\Rightarrow$ $I(n)$ is reachable from m_2

Similarly, $\forall n \in N_2$, n is reachable from m_2 by a shortest path of length 1

$\Rightarrow$ $\exists i: I(r_i) = n$ [(3)] $\Rightarrow$ $I^{-1}(n) = r_i$ is reachable from m_1 [(2)+(3)].

Induction step: Assume that A is true for any node reachable from m_1

(or m_2) by a shortest path of length k > 0.

$\forall n \in N_1$, n is reachable from m_1 by a shortest path of length k+1 $\Rightarrow$

$\exists n' \in N_1$: n is a successor of n' and n' is reachable from m_1 by a shortest

path of length k. n' is reachable from m_1 by a shortest path of length

k $\Rightarrow$ $I(n')$ is reachable from m_2 [ind. hyp.]. n is a successor of n' $\Rightarrow$

$I(n)$ is a successor of $I(n')$ [(4)]. Therefore, n_1 is reachable from m_1

by a shortest path of length k+1 $\Rightarrow$ $I(n')$ is reachable from m_2 and $I(n)$

is a successor of $I(n')$ $\Rightarrow$ $I(n)$ is reachable from m_2. As in the basis

for A, a symmetric argument will show that $\forall n \in N_2$, n is reachable from m_2

by a shortest path of length k+1 $\Rightarrow$ $I^{-1}(n)$ is reachable from m_1. Thus

A is proven by induction.

(8) Let n_2 be m_2 or any node reachable from m_2 in U_2. Then $I^{-1}(n_2)$ is

reachable from m_1 in U_1 A

$SM_2(I(I^{-1}(n_2))) = I(SM_1(I^{-1}(n_2)))$ [(2)]; i.e., $SM_2(n_2) = I(SM_1(I^{-1}(n_2)))$

[(1)], so $SM_1(I^{-1}(n_2)) = \{v, (s_1, r_1), \ldots, (s_j, r_j)\}$ iff

$SM_2(n_2) = \{v, (s_1, I(r_1)), \ldots, (s_j, I(r_j))\}$. Furthermore,

$SM_1(I^{-1}(n_2)) = \{v, (s_1, r_1), \ldots, (s_j, r_j)\}$ iff

$SM_1(I^{-1}(n_2)) = \{v, (s_1, I^{-1}(I(r_1))), \ldots, (s_j, I^{-1}(I(r_j)))\}$ [(1)]. Therefore

(9) $SM_1(I^{-1}(n_2)) = I^{-1}(SM_2(n_2))$

Since $m_1 = I^{-1}(m_2)$ [(1)+(2)],

(10) $U_2 \cdot m_2 \overset{I}{=} U_1 \cdot m_1 \to U_1 \cdot m_1 \overset{I^{-1}}{=} U_2 \cdot m_2$ (2)+(8)+(9)

Now, $Match((b_2, S_2), I, (b_1, S_1))$

     $\to$ b_1 has no token in Γ_1 and b_2 has no token in Γ_2, or

       b_1 has a non-pointer value in Γ_1 and b_2 has the same value in Γ_2, or

       b_1 has a pointer value p_1, b_2 has a pointer value p_2, and

$$U_2 \cdot \Pi_2(p_2) \overset{I}{=} U_1 \cdot \Pi_1(p_1)$$

     $\to$ b_2 has no token in Γ_2 and b_1 has no token in Γ_1, or

       b_2 has a non-pointer value in Γ_2 and b_1 has the same value in Γ_1, or

       b_2 has a pointer value p_2, b_1 has a pointer value p_1, and

$$U_1 \cdot \Pi_1(p_1) \overset{I^{-1}}{=} U_2 \cdot \Pi_2(p_2) \quad [(10)]$$

$\to Match((b_1, S_1), I^{-1}, (b_2, S_2))$

Transitivity: Let $S_i = (\Gamma_i, U_i)$, for i=1,2,3, be any three states, where $U_i = (N_i, \Pi_i, SM_i)$. Let b_i be any arc from the program of which Γ_i is a configuration. Then prove that for any two one-to-one mappings $I_1: N_1 \to N_2$ and $I_2: N_2 \to N_3$, $Match((b_2, S_2), I_1, (b_1, S_1))$ and $Match((b_3, S_3), I_2, (b_2, S_2)) \to Match((b_3, S_3), I_2 \cdot I_1, (b_1, S_1))$, where $I_2 \cdot I_1$ is the composition of the mappings I_1 and I_2.

(11) Let $m_1 \in N_1$, $m_2 \in N_2$, and $m_3 \in N_3$ be any three nodes for which $U_2 \cdot m_2 \overset{I_1}{=} U_1 \cdot m_1$ and $U_3 \cdot m_3 \overset{I_2}{=} U_2 \cdot m_2$. Then $m_2 = I_1(m_1)$, $m_3 = I_2(m_2)$, for each node n_1 equal to or reachable from m_1, $SM_2(I_1(n_1)) = I_1(SM_1(n_1))$, and for any node n_2 equal to or reachable from m_2, $SM_3(I_2(n_2)) = I_2(SM_2(n_2))$.

(12) $m_3 = I_2 \cdot I_1(m_1)$ (11)

(13) Let n_1 be any node equal to or reachable from m_1. Then $I_1(n_1) \in N_2$

     is equal to or reachable from $I_1(m_1) = m_2$ (11)+A

$SM_1(n_1) = \{v, (s_1, r_1), \ldots, (s_j, r_j)\}$ iff

$SM_2(I_1(n_1)) = \{v, (s_1, I_1(r_1)), \ldots, (s_j, I_1(r_j))\}$ $[(11)]$ iff

$SM_3(I_2(I_1(n_1))) = \{v, (s_1, I_2(I_1(r_1))), \ldots, (s_j, I_2(I_1(r_j)))\}$ [(13)+(11)].

I,e., $SM_3(I_2 \cdot I_1(n_1)) = I_2 \cdot I_1(SM_1(n_1))$. Therefore,

(14) $U_3.m_3 \overset{I_2 \cdot I_1}{=} U_1.m_1$ (12)+(13)

$\text{Match}((b_2, S_2), I_1, (b_1, S_1)) \Rightarrow$

   $[b_1$ has no token in $\Gamma_2 \Rightarrow b_2$ has no token in $\Gamma_2] \wedge$

   $[b_1$ has a non-pointer value $\Rightarrow b_2$ has the same value$] \wedge$

   $[b_1$ has pointer value $p_1 \Rightarrow b_2$ has a pointer value p_2 such that
       $U_2.\Pi_2(p_2) \overset{I_1}{=} U_1.\Pi_1(p_1)]$,

and $\text{Match}((b_3, S_3), I_2, (b_2, S_2)) \Rightarrow$

   $[b_2$ has no token in $\Gamma_2 \Rightarrow b_3$ has no token in $\Gamma_3] \wedge$

   $[b_2$ has a non-pointer value $\Rightarrow b_3$ has the same value$] \wedge$

   $[b_2$ has pointer value $p_2 \Rightarrow b_3$ has a pointer value p_3 such that
       $U_3.\Pi_3(p_3) \overset{I_2}{=} U_2.\Pi_2(p_2)]$,

so $\text{Match}((b_2, S_2), I_1, (b_1, S_1))$ and $\text{Match}((b_3, S_3), I_2, (b_2, S_2)) \Rightarrow$

   $[b_1$ has no token in $\Gamma_1 \Rightarrow b_3$ has no token in $\Gamma_3] \wedge$

   $[b_1$ has a non-pointer value $\Rightarrow b_3$ has the same value$] \wedge$

   $[b_1$ has pointer value $p_1 \Rightarrow b_3$ has a pointer value p_3 such that
       $U_3.\Pi_3(p_3) \overset{I_2 \cdot I_1}{=} U_1.\Pi_1(p_1)]$ [(14)]

$\Rightarrow \text{Match}((b_3, S_3), I_2 \cdot I_1, (b_1, S_1))$

$\triangle$

Appendix B

Proof of Theorem 3.4-2

Theorem 3.4-2 Let P be any well-behaved L_{BV} program, and let P' be its translation via Algorithm 3.4-1. Let S be any initial standard state for P, and let S' be any initial modified state for P' which simulates S. Then for any halted firing sequence Ω starting in S:

1. $R(\Omega)$ is a halted firing sequence starting in S', and

2. $S' \cdot R(\Omega)$ simulates $S \cdot \Omega$.

Proof:

Key definitions: Def. 2.4-1 - equal components; Def. 3.4-1 - Match;

Defs. 2.1-5+2.2-5 - standard interpreter;

Defs. 3.3-7+3.3-8+3.3-9 - modified interpreter

Proof is by induction on the lengths of the prefixes of Ω. For any prefix θ, let $\theta' = R(\theta)$, and let the state $S' \cdot \theta'$ be (Γ', U', Q'). Let A and T be the maps generated by Algorithm 3.4-1 in translating P into P'. Then the induction hypotheses are:

V: $R(\theta)$ is a firing sequence starting in S'.

W: There is no write pointer on any arc in Γ'.

X: Q' is empty.

Y: For any arc b' which is a number-1 input arc of an Assign, Update, or Delete, or of a sequencer in P', there is no token on b' in Γ'.

Z: $S' \cdot \theta'$ simulates $S \cdot \theta$.

Basis: $|\theta| = 0$. Then $\theta' = \theta = \lambda$ [Alg. 3.4-2], so

(1) θ' is a firing sequence starting in S', $S' \cdot \theta' = S'$, and $S \cdot \theta = S$

There are no write pointers in an initial modified state, and the pool component therein is empty [Def. 3.3-5]. The only arcs in P' which have tokens on them in an initial state are program input arcs and control arcs [(1)+Defs. 3.3-5+2.2-6], and none of those arcs is a number-1 input arc of an Assign, Update, Delete, or sequencer. Since S' simulates S by hypothesis, $S' \cdot \theta'$ simulates $S \cdot \theta$ [(1)]. Hence, V, W, X, Y, and Z for θ. Induction step: Assume that the five induction hypotheses are true for some proper prefix θ of Ω. Consider prefix $\theta\varphi$ of Ω, in which the last firing φ is of an actor labelled d in P. Use the following notation:

(Γ_1, U_1) is the state $S \cdot \theta$, where $U_1 = (N_1, \Pi_1, SM_1)$

(Γ_2, U_2) is the state $S \cdot \theta\varphi$, where $U_2 = (N_2, \Pi_2, SM_2)$

(Γ_1', U_1', Q_1') is the state $S' \cdot \theta'$, where $U_1' = (N_1', \Pi_1', SM_1')$

(Γ_2', U_2', Q_2') is the state $S' \cdot R(\theta\varphi)$, where $U_2' = (N_2', \Pi_2', SM_2')$

(2) d is enabled in Γ_1 Def. 2.3-1

(3) There is a mapping $I: N_1 \to N_1'$ under which, for any arc b in P,

Match$((A(b), S' \cdot \theta'), I, (b, S \cdot \theta))$ ind. hyp. Z+Def. 2.4-7

(4) For any arc b in P, there is a token on b in Γ_1 iff there is a token on A(b) in Γ_2 (3)

There are two cases to consider: d either is or is not a Const, Append, or Remove.

Case I: d is not a Const, Append, or Remove.

(5) T(d) is the label of a single actor in P', having the same type as d, and for each input and output arc b of d in P, A(b) is the same input or output arc of T(d) in P'

T(d) is enabled in Γ_1', unless it is a Select S and there is a pointer p

such that $S \in Q_1^!(p)$ [(2)+(4)+(5)+Def. 3.3-6], so

(6) T(d) is enabled in $\Gamma_1^!$ ind. hyp. X

Since P is an L_{BV} program, d is not a Copy, Assign, Update, or Delete

[Def. 2.2-3], so

(7) T(d) is not a Copy, Assign, Update, or Delete   (5)

(8) $\theta'\varphi'$, where φ' is the firing which is the label T(d), is a firing

      sequence starting in S' ind. hyp. V+(6)+(7)+Def. 2.3-1

Since $\theta'\varphi' = R(\theta)\varphi'$ is $R(\theta\varphi)$ [(8)+(7)+Alg. 3.4-2], V for $\theta\varphi$ [(8)].

Let (Γ'', U'', Q'') be the state Fire$(S' \cdot \theta', T(d))$, and let $\Gamma_s^!$ be

Standard$_\Gamma((\text{Strip}(\Gamma_1^!, T(d)), U_1), T(d))$. Then there is a write pointer on

an arc b in $\Gamma_2^!$ $\Rightarrow$ there is a write pointer on b in Γ'' $\Rightarrow$ since T(d) is

not a Copy [(7)], there is a write pointer on b in $\Gamma_s^!$ $\Rightarrow$ there is a write

pointer on b in $\Gamma_1^!$, unless b is an output arc of pI actor T(d), in which

case there is a write pointer on an input arc of T(d) in $\Gamma_1^!$. Therefore,

(9) W for $\theta\varphi$, and there is no write pointer on b in Γ'' ind. hyp. W

T(d) is not a Select with a pointer on any output arc in $\Gamma_1^!$ $\Rightarrow$

$\Gamma'' = \Gamma_s^!$ and Q'' is empty [(7)+ind. hyp. X] $\Rightarrow$ $\Gamma_2^! = \Gamma'' = \Gamma_s^!$ and $Q_2^!$ is empty.

T(d) is a Select with pointer p on its output arcs in $\Gamma_s^!$ $\Rightarrow$ Γ'' is $\Gamma_s^!$

with all those output tokens removed, and Q'' is empty except that Q''(p) is

{T(d)} [ind. hyp. X] $\Rightarrow$ $\Gamma_2^!$ is $\Gamma_s^!$ with tokens on the output arcs of T(d)

whose value is (p,R), and $Q_2^!$ is empty [(9)]. Therefore,

(10) X for $\theta\varphi$, and $\Gamma_2^!$ equals $\Gamma_s^!$ with "R" tags in each pointer-valued

      token on an output arc of T(d)

For any arc b in P', there is a token on b in $\Gamma_s^!$ but not in $\Gamma_1^!$ only

if b is an output arc of T(d). Every number-1 input arc of an Assign,

Update, Delete, or sequencer is an output arc of a Copy, Assign, Update,

or Delete. Hence, Y for $\theta\varphi$ [(10)+(7)+ind. hyp. Y].

$$U_2' = \text{Standard}_U((\text{Strip}(\Gamma_1', T(d)), U_1'), T(d)), \text{ so}$$

(11) $U_2 = U_1$ and $U_2' = U_1'$ (8)+(7)

(12) For every node m in $N_2 = N_1$, $U_2 \cdot m \overset{M}{=} U_1 \cdot m$, and for every node m'

    in $N_2' = N_1'$, $U_2' \cdot m' \overset{M}{=} U_1' \cdot m'$, where M is the identity mapping   (11)

Notation: For any two configurations Γ and Γ', any two arcs b and b' in

the programs of which Γ and Γ' are configurations, and any value $v \in V$,

abbreviate "there is a token of value v on b in Γ iff there is a token

of value v on b' in Γ'" by "$TV(b, \Gamma) = TV(b', \Gamma') = v$".

(13) For every arc b in P, b is not an input or output arc of d $\Rightarrow$

    $TV(b, \Gamma_2) = TV(b, \Gamma_1) \wedge A(b)$ is not an input or output arc of $T(d)$

    $[(5)] \Rightarrow TV(A(b), \Gamma_2') = TV(A(b), \Gamma_1')$ (10)

(14) d is a gate and c is its control input arc $\Rightarrow T(d)$ is the same type

    of gate, and A(c) is its control input arc $[(5)] \Rightarrow T(d)$ has the

    same control input in Γ_1' as d has in Γ_1 (3)

For any arc b in P, b is an input arc of d $\Rightarrow$ there is a token on b in Γ_2

iff there is a token on A(b) in Γ_s', hence in Γ_2', and if so,

$TV(b, \Gamma_2) = TV(b, \Gamma_1)$ and $TV(A(b), \Gamma_2') = TV(A(b), \Gamma_1')$ [(8)+(5)+(14)+(10)].

Thus,

(15) For any arc b in P, b is not an output arc of d $\Rightarrow$ there is a token

    on b in Γ_2 iff there is a token on A(b) in Γ_2', and if so,

    $TV(b, \Gamma_2) = TV(b, \Gamma_1)$ and $TV(A(b), \Gamma_2') = TV(A(b), \Gamma_1')$ [(13)] $\Rightarrow$ there

    is a token on b in Γ_2 iff there is a token on A(b) in Γ_2', and if

    so, Match$((b, S \cdot \theta\varphi), M, (b, S \cdot \theta))$ and Match$((A(b), S' \cdot \theta'\varphi'), M,$

    $(A(b), S' \cdot \theta'))$ [(12)] $\Rightarrow$ Match$((A(b), S' \cdot \theta'\varphi'), I, (b, S \cdot \theta\varphi))$

                                                             (3)+Thm. 2.4-1

For the output arcs of d, there are three subcases to consider.

Case Ia: d is a pI actor. Then there are tokens on all output arcs of d in Γ_2 iff there are tokens on all output arcs of T(d) in Γ'_s, hence in Γ'_2, and if there are such tokens, then there is an arc a in P such that, for any output arc b of d, $TV(b,\Gamma_2) = TV(a,\Gamma_1)$ and $TV(A(b),\Gamma'_2) = TV(A(a),\Gamma'_1)$ [(8)+(5)+(14)]. Hence, $Match((b,S \cdot \theta\varphi), M, (a,S \cdot \theta))$ and $Match((A(b),S' \cdot \theta'\varphi'), M, (A(a),S' \cdot \theta'))$ [(12)], so

(16) $Match((A(b),S' \cdot \theta'\varphi'), I, (b,S \cdot \theta\varphi))$ (3)+Thm. 2.4-1

Case Ib: d is neither a pI nor a structure operator

(17) For each input arc a of d in P, there is a non-pointer value on a in Γ_1, and there is a non-pointer value on each output arc of d in Γ_2

By (3), then, $TV(A(a),\Gamma'_1) = TV(a,\Gamma_1)$. For each output arc b of d, there is a token on b in Γ_2 and one on A(b) in Γ'_s, and the value of the token on A(b) in Γ'_s depends on just the tokens on T(d)'s input arcs in Γ'_1, in exactly the same way that the value of the token on b in Γ_2 depends just on the values of the tokens on d's input arcs in Γ_1 [(8)+(5)]. Therefore, $TV(A(b),\Gamma'_2) = TV(b,\Gamma_2)$ [(10)], so

(18) $Match((A(b),S' \cdot \theta'\varphi'), I, (b,S \cdot \theta\varphi))$

Case Ic: d is a structure operator

(19) d is a Fetch, First, Next, or Select   Def. 2.2-3

(20) If d is a Next or Select, then it has a selector input arc a, T(d) has a selector input arc A(a), and $TV(a,\Gamma_1) = TV(A(a),\Gamma'_1)$ (5)+(3)

(21) Let p be the value on d's pointer input arc in Γ_1, and let $m = \Pi_1(p)$. Then there is a token with pointer value p', (p',R), or (p',W) on the pointer input arc of T(d) in Γ'_1, and, letting $m' = \Pi'_1(p')$,

$$U_1' \cdot m' \overset{I}{=} U_1 \cdot m \qquad \qquad (5)+(3)$$

(22) $SM_1(m) = \{v, (s_1, n_1), \ldots, (s_j, n_j)\}$ iff

$SM_1'(m') = \{v, (s_1, I(n_1)), \ldots, (s_j, I(n_j))\}$

(23) Let b be any output arc of d. Then A(b) is the same output arc of

T(d) $\qquad \qquad (5)$

d is a Fetch $\Rightarrow$ $TV(b, \Gamma_2)$ depends only on the value in $SM_1(m)$, and

T(d) is a Fetch, so $TV(A(b), \Gamma_s')$ depends in the same way on the value in

$SM_1'(m')$ $[(5)+(21)+(23)]$ $\Rightarrow$ $TV(A(b), \Gamma_2') = TV(b, \Gamma_2)$ $[(22)+(10)]$.

d is a First or Next or d is a Select and b is its control output arc

$\Rightarrow$ $TV(b, \Gamma_2)$ depends only on the set of selectors in $SM_1(m)$ and on the

value of the token on d's selector input arc a, if any, and since T(d)

is the same type of actor, the value on A(b) in Γ_s' depends in the same

way on the set of selectors in $SM_1'(m')$ and on the value of the token on

T(d)'s selector input arc A(a), if any $[(5)+(21)+(23)]$ $\Rightarrow$ $TV(b, \Gamma_2) =$

$TV(A(b), \Gamma_2')$ $[(22)+(20)+(10)]$. From these two paragraphs,

(24) d is a Fetch, First, or Next, or b is a control output arc of a

Select d $\Rightarrow$ the value of the token on b in Γ_2 is not a pointer,

and $TV(b, \Gamma_2) = TV(A(b), \Gamma_2')$ $\Rightarrow$ Match$((A(b), S' \cdot \theta' \varphi'), I, (b, S \cdot \theta \varphi))$

Otherwise, d is a Select and b is a data output arc of it. Let s

be the value of the token on d's selector input arc in Γ_1. Then T(d) is

a Select with a selector input of s $[(19)+(24)+(5)+(20)]$. $\nexists i: s_i = s$

$\Rightarrow$ the value of the token on b in Γ_2 and of that on A(b) in Γ_s' are both

undef $[(22)]$. $\exists i: s_i = s$ $\Rightarrow$ the value of the token on b in Γ_2 is q,

where $\Pi_1(q) = n_1$, and the value of the token on A(b) in Γ_s' is q', where

$\Pi_1'(q') = I(n_1)$ $[(22)]$ $\Rightarrow$ the value of the token on A(b) in Γ_2' is (q',R)

$[(10)]$. In this latter case, $\Pi_2'(q') = I(n_1) = I(\Pi_2(q))$ $[(11)]$, and for

any node n, n is equal to or reachable from $n_1 = \Pi_2(q)$ in $U_2 \Rightarrow$ n is

reachable from m in U_2, hence in U_1 $[(22)+(11)+$Def. 2.2-2$] \Rightarrow SM_2'(I(n)) =$

$SM_1'(I(n)) = I(SM_1(n)) = I(SM_2(n))$ $[(11)+(21)]$. By Def. 2.4-1, then,

$U_2' \cdot \Pi_2'(q') \stackrel{I}{=} U_2 \cdot \Pi_2(q)$. Therefore, in either case,

(25) d is a Select and b is a data output arc $\Rightarrow$

   $Match((A(b), S' \cdot \theta' \varphi'), I, (b, S \cdot \theta \varphi))$.

Hence, Z for $\theta \varphi$ $[(15)+(16)+(18)+(19)+(24)+(25)+$Def. 2.4-7$]$

Case II: d is a Const, Append, or Remove

(26) Let T(d) be the triple (C,U,G). Then in P', C labels a Copy, G

   a sequencer, and U either an Assign, Update, or Delete

(27) There are tokens on all of d's input arcs and on none of its output

   arcs in Γ_1, so   (2)+Def. 2.1-4

(28) There are tokens on C's input arc and on U's number-2 (and number-3)

   input arcs in Γ_1' (4)

For every output arc b of C, U, or G in P', either b is an input arc of

an Assign, Update, or Delete, or a sequencer, or there is an output arc

a of d in P such that b = A(a), so

(29) No output arc of C, U, or G holds a token in Γ_1' (27)+Def. 2.1-4

C is enabled in Γ_1' $[(28)+(29)]$, so

(30) $\theta' \varphi_C$, where $\varphi_C = (C,(p,n))$, $p \notin$ dom Π_1' and $n \notin N_1'$, is a firing sequence

   starting in S' ind. hyp. W+Def. 2.3-1

There is a token on U's number-1 input arc in $S' \cdot \theta' \varphi_C$, so U is enabled

$[(28)+(29)+(30)]$, and so

(31) $\theta' \varphi_C \varphi_U$, where $\varphi_U = U$, is a firing sequence starting in S' (30)

There are tokens on both of G's input arcs in $S' \cdot \theta' \varphi_C \varphi_U$, so G is enabled

$[(29)+(30)+(31)]$, and so

(32) $\theta' \varphi_C \varphi_U \varphi_G$, where $\varphi_G = G$, is a firing sequence starting in S' (31)

(33) $R(\theta\varphi) = R(\theta) \varphi_C \varphi_U \varphi_G$ is a firing sequence starting in S'

                                      (30)+(31)+(32)+Alg. 3.4-2

The only number-1 output arc of C is an input arc of U, so the only write pointer output by any of the firings φ_C, φ_U, or φ_G is input by φ_U; i.e., W for $\theta\varphi$ [(33)+ind. hyp. W].

Since none of φ_C, φ_U, or φ_G is a Select firing, X for $\theta\varphi$ [(33)+(26)+ ind. hyp. X].

For every number-1 input arc of an Assign, Update, or Delete, or sequencer in P' on which a token is placed by one of φ_C, φ_U, or φ_G in $R(\theta\varphi)$, that token is removed by a subsequent one of those firings. Hence Y for $\theta\varphi$ [ind. hyp. Y].

(34) Let a be the number-1 input arc of d. Let p_1 (p_1') be the value of the token on arc a (A(a)) in Γ_1 (Γ_1'). Let $m_1 = \Pi_1(p_1)$ and $m_1' = \Pi_1'(p_1')$. Then $U_1' \cdot m_1' \overset{I}{=} U_1 \cdot m_1$ (3)

(35) $SM_1(m_1) = \{v, (s_1,n_1),\ldots,(s_j,n_j)\}$ iff
$SM_1'(m_1') = \{v, (s_1,I(n_1)),\ldots,(s_j,I(n_j))\}$ (34)

(36) Let p_2' be the value of the token on C's output arcs in $S' \cdot \theta' \varphi_C$, and let $m_2' = \Pi_2'(p_2')$. Then p_2' was output by φ_C, and letting the heap in $S' \cdot \theta' \varphi_C$ be (N_3',Π_3',SM_3'), $SM_3'(m_2') = SM_1'(m_1')$ (34)+(30)

(37) p_2' is the number-1 input to φ_U and the transmitted input of φ_G

                                        (36)+(31)+(32)

(38) Let p_2 be the value of the token on d's data output arcs in Γ_2, and let $m_2 = \Pi_2(p_2)$. Define I^+ to be $IU\{(m_2,m_2')\}$. Then $SM_3'(m_2') = \{v, (s_1,I^+(n_1)),\ldots,(s_j,I^+(n_j))\}$.   (36)+(35)

(39) For any control output arc b of d, A(b) is a control output arc of U

The value of the token on b in $S \cdot \theta \varphi$ depends on the value and set of selectors in $SM_1(m_1)$ [(34)], and the value of the token on A(b) in $S' \cdot \theta' \varphi_C \varphi_U$, hence in $S' \cdot \theta' \varphi_C \varphi_U \varphi_G$, depends in the same way on $SM_3'(\Pi_2'(p_2')) = SM_3'(m_2')$ [(26)+(39)+(37)]. Therefore,

(40) For every control output arc b of d, $TV(b, \Gamma_2) = TV(A(b), \Gamma_2')$ (35)+(38)

Every output arc of G holds a token of value p_2' in Γ_2' [(37)+ Def. 3.2-2]. Every output arc of C which is not an input arc of U or G holds a token of value p_2' in Γ_2' [(36)+(33)]. For every data output arc b of d, A(b) is either an output arc of G or an output arc of C which is not an input arc of U or G [(26)]. Therefore,

(41) b holds a token of value p_2 in Γ_2, A(b) holds a token of value p_2'

in Γ_2', and $\Pi_2'(p_2') = I^+(\Pi_2(p_2))$ (38)

(42) $\Pi_1 \subset \Pi_2$ and $\forall n \in N_2$, $n \neq m_2 \Rightarrow SM_2(n) = SM_1(n)$ (38)

(43) The only firing among φ_C, φ_U, and φ_G which changes SM is φ_U, so

$\Pi_1' \subset \Pi_2'$, and $\forall n \in N_2'$, $n \neq m_2' \Rightarrow SM_2'(n) = SM_1'(n)$ (37)+(36)

(44) $SM_2(m_2)$ is created from $SM_1(m_1)$ by the firing φ of d, and $SM_2'(m_2')$

is created from $SM_3'(m_2')$ by the firing φ_U of U   (38)+(37)+(36)

Letting a_2 (a_3) be the number-2 (number-3) input arc of d, $A(a_2)$ $(A(a_3))$ is the number-2 (number-3) input arc of U [(26)], so

(45) φ and φ_U have equal number-2 inputs, and for their number-3 inputs,

p_3 and p_3', $U_1' \cdot \Pi_1'(p_3') \overset{I}{=} U_1 \cdot \Pi_1(p_3)$ (3)

(46) d is a Const $\Rightarrow SM_2(m_2) = \{v', (s_1, n_1), \dots, (s_j, n_j)\}$, where v' is

φ's number-2 input [(44)] $\wedge$ φ_U is an Assign firing with v' as its

number-2 input [(26)+(45)] $\Rightarrow$

$SM_2'(m_2') = \{v', (s_1, I^+(n_1)), \dots, (s_j, I^+(n_j))\}$ [(44)]

(47) d is a Remove $\Rightarrow SM_2(m_2) = SM_1(m_1) - \{(s_1, n_1)\}$, where s_1 is φ's

number-2 input $[(44)] \wedge \varphi_U$ is a Delete firing with s_1 as its

number-2 input $[(26)+(45)] \Rightarrow SM_2'(m_2') = SM_3'(m_2') - \{(s_1, I^+(n_1))\}$

$$(38)+(44)$$

(48) d is an Append $\Rightarrow SM_2(m_2) = SM_1(m_1) \cup \{(s, \Pi_1(p_3))\}$, where s (p_3) is

φ's number-2 (number-3) input $[(44)] \wedge U$ is an Update with number-2

input s and number-3 input p_3' where $U_1' \cdot \Pi_1'(p_3') \overset{I}{=} U_1 \cdot \Pi_1(p_3)$ $[(26)+$

$(45)] \Rightarrow SM_2'(m_2') = SM_3'(m_2') \cup \{(s, \Pi_1'(p_3'))\}$ $[(42)+(43)+(44)] \Rightarrow$

$SM_2'(m_2') = SM_3'(m_2') \cup \{(s, I^+(\Pi_2(p_3)))\}$ $[(38)]$

(49) $SM_2'(m_2') = I^+(SM_2(m_2))$ $\qquad\qquad (35)+(38)+(46)+(47)+(48)$

(50) Let b be any arc of P. b is an input arc of $d \Rightarrow$ b is empty in Γ_2

$\wedge A(b)$ is an input arc of C or U in P' $[(26)] \Rightarrow A(b)$ is empty in

Γ_2' $[(33)]$

(51) b has a token in Γ_2 and is not an output arc of $d \Rightarrow$

$TV(b, \Gamma_2) = TV(b, \Gamma_1) \wedge A(b)$ is not an output arc of C, U, or G

$[(26)] \Rightarrow TV(A(b), \Gamma_2') = TV(A(b), \Gamma_1')$ $[(33)]$

b has a token of non-pointer value in $\Gamma_1 \Rightarrow TV(b, \Gamma_1) = TV(A(b), \Gamma_1')$ $[(3)]$,

so

(52) b has a token of non-pointer value in $\Gamma_2 \Rightarrow TV(b, \Gamma_2) = TV(A(b), \Gamma_2')$

$$(40)+(51)$$

(53) Let b be any arc which holds a pointer in Γ_2, and let p be that

pointer. b is not an output arc of $d \Rightarrow$ b holds a token of value

p in Γ_1, $A(b)$ holds a token of value p' in Γ_1' and Γ_2', and

$U_1' \cdot \Pi_1'(p') \overset{I}{=} U_1 \cdot \Pi_1(p) \Rightarrow \Pi_2'(p') = I^+(\Pi_2(p))$ $\quad (51)+(3)+(38)+(42)+(43)$

(54) b holds a token of value p in Γ_2, $A(b)$ holds a token of value p'

in Γ_2', and $\Pi_2'(p') = I^+(\Pi_2(p))$ $\qquad\qquad (41)+(53)$

(55) Let n be any node equal to or reachable from $\Pi_2(p)$. $n = m_2 \Rightarrow$

$SM_2'(I^+(n)) = I^+(SM_2(n))$ (49)+(38)

m_2 is not in N_1, so for any $n \in N_1$, there is no s such that (s, m_2) is in

$SM_1(n)$ [(38)+Thm. 2.2-1]; i.e., m_2 is not reachable from any node in U_2,

that is, every path containing m_2 in U_2 starts at m_2 [Def. 2.2-2]. Thus,

(56) Every path in U_2 not starting at m_2 is a path in U_1 (42)+Def. 2.2-2

(57) $n \neq m_2$ and b is not an output arc of $d \Rightarrow p \neq p_2 \wedge U_1'.\Pi_1'(p') \overset{I}{=} U_1.\Pi_1(p)$

$\Rightarrow$ n is equal to or reachable from $\Pi_2(p)$ in U_2, hence is equal

to or reachable from $\Pi_1(p)$ in $U_1 \Rightarrow SM_1'(I(n)) = I(SM_1(n))$

                                          (53)+(38)+(56)

(58) $n \neq m_2$ and b is an output arc of $d \Rightarrow p = p_2 \Rightarrow$ n is reachable from

$m_2 = \Pi_2(p_2)$ in $U_2 \Rightarrow$ n is reachable from $m_1 = \Pi_1(p_1)$ or possibly n

equals or is reachable from $\Pi_2(p_3)$ in U_2

                               (38)+(46)+(47)+(48)+Def. 2.2-2

$n \neq m_2 \Rightarrow SM_2'(I^+(n)) = I^+(SM_2(n))$ [(57)+(58)+(43)+(38)+(42)], so for

any node n equal to or reachable from $\Pi_2(p)$ in U_2,

$SM_2'(I^+(n)) = I^+(SM_2(n))$ [(55)]. Therefore, $U_2'.\Pi_2'(p') \overset{I^+}{=} U_2.\Pi_2(p)$ [(54)],

and so for any arc b in P, $Match((A(b), S' \cdot R(\theta\varphi)), I^+, (b, S \cdot \theta\varphi))$

[(50)+(52)+(53)+(54)]. Hence, Z for $\theta\varphi$ [Def. 2.4-7].

Thus it is proven inductively that

(59) $\Omega' = R(\Omega)$ is a firing sequence starting in S',

(60) there is no token on any number-1 input arc of an Assign, Update,

Delete, or sequencer in $S' \cdot \Omega'$, and

(61) $S' \cdot \Omega'$ simulates $S \cdot \Omega$.

(62) No actor is enabled in $S \cdot \Omega$ Def. 2.3-1

Ω' is not halted $\Rightarrow$ there is some actor d' which is enabled in $S' \cdot \Omega'$

[Def. 2.3-1] $\Rightarrow$ [d' is not a Copy, Update, Delete, or sequencer $\Rightarrow$ there

is an actor labelled d in P such that T(d) = d' $\Rightarrow$ d is enabled [(5)+(3)+

(4)+Def. 2.1-4]] $\Rightarrow$ d' is a Copy, Assign, Update, Delete or sequencer

[(62)] $\Rightarrow$ letting (C,U,G) be the triple in the range of T containing d',

U and G are not enabled [(60)+Def. 2.1-4] $\Rightarrow$ C is enabled $\Rightarrow$ there is a

token on C's input arc in $S' \cdot \Omega'$ $\Rightarrow$ for some input arc b of d, where

T(d) = (C,U,G), there is a token on A(b) in P' $\Rightarrow$ there is a token on an

input arc of d in $S \cdot \Omega$ [(61)+Def. 2.4-7] $\Rightarrow$ there is in $S \cdot \Omega$ a token on

an arc which is not a program output arc or a control arc [Def. 2.1-1]

$\Rightarrow$ P is not well-behaved [Def. 2.3-2]. Therefore, $R(\Omega)$ is halted.

Q.E.D.

Appendix C

Proof of Lemma 4.3-2

Lemma 4.3-2 Let S be any initial standard or modified state of any program P, and let Ω be any firing sequence starting in S. Then $\eta(S,\Omega)$ is a causal computation for Int(P).

Proof:

Key definitions: Def. 4.2-7 - causality;   Def. 4.3-1 - DL, In;

         Def. 4.3-2 - Int(P) = (St, l, IE); Alg. 4.3-1 - $\eta(S,\Omega)$

Prove first that $\omega(S,\Omega)$ is causal and that the destinations of the transfers in $\omega(S,\Omega)$ are all distinct; do so by induction on $|\Omega|$.

Basis: $|\Omega| = 0$. Then $\omega(S,\Omega) = \lambda$, so there are no entries in $\omega(S,\Omega)$. Furthermore, no execution has output entries in $\omega(S,\Omega)$, so $\omega(S,\Omega)$ is causal.

Induction step: Assume that $\omega(S,\Omega)$ is causal and that the transfers of the entries in it all have distinct destinations for any Ω of length $m \geq 0$. Consider $\theta\varphi$ of length $m+1$, in which the last firing φ is of the actor labelled d. Let $\alpha = \omega(S,\theta)$ and $\beta = \omega(S,\theta\varphi)$. (All initiations are with respect to Int(P).)

(1) Let Ex(d,n) be any execution of which there is an input or an output entry in β. Then either d is the label of an actor in P or d$\in$DL, so d$\in$St   Def. 4.2-5

(2) α is a prefix of β

(3) Let f be any entry in β, and let it be an output entry of

e = Ex(d,k). k = 0 $\Rightarrow$ d$\in\{$"ID","IT","IF"$\}$ $\Rightarrow$ In(l(d)) = 0 $\Rightarrow$ e is

initiated in any computation   Def. 4.2-6

$f \in \alpha \Rightarrow$ e's initiating entry precedes f in α, hence in β [(2)+ind. hyp.].

f is in β but not in α and $k > 0 \Rightarrow$ the source in T(f) is Source(b,S,θ)

for some arc b $\Rightarrow$ d$\in$St-DL and there are exactly k firings of d in θ

[(1)+(2)] $\Rightarrow$ there are In(I(d)) input entries to e in $\omega(S,\theta) = \alpha$

[Lemma 4.3-1] $\Rightarrow$ e's initiating entry is in α, hence it precedes f in β

[(2)+Def. 4.2-6]. Therefore, β is causal.

Let n be such that φ is the nth firing of actor c in $\theta\varphi$. Then there

is exactly one entry in β which is not in α for each token removed from

an input arc of c in the transition from $S \cdot \theta$ to $S \cdot \theta\varphi$, and the destination

in the transfer of each such entry is Dst(Ex(c,n),j), where the token

was removed from c's number-j input arc, and c$\in$St-DL [(1)]. Since there

are fewer than n firings of c in θ, there are 0 input entries to Ex(c,n)

in α; i.e., none of the entries in α has Ex(c,n) in the destination of

its transfer [Lemma 4.3-1]. Since

(4) for any actor c, for each j, there is at most one number-j input

  arc to c, and at most one token is removed from it in any

  transition   Defs. 2.1-1+2.1-5

the transfers of the entries in β-α have distinct destinations, although

each of them has Ex(c,n) in it. By induction hypothesis, the destinations

of the transfers of the entries in α are all distinct. Therefore, the

destinations of the transfers of the entries in β are all distinct.

Thus it is proven by induction that

(5) for any firing sequence Ω, $\omega(S,\Omega)$ is causal and the destinations in

  the transfers of all entries in $\omega(S,\Omega)$ are all distinct.

(6) If Ω is not halted, then $\eta(S,\Omega) = \omega(S,\Omega)$

(7) Assume Ω is halted, and let $\alpha = \omega(S,\Omega)$ and $\beta = \eta(S,\Omega)$. Let f be

any entry in β and let it be an output entry of $e = Ex(c,n)$.

$\quad n = 0 \Rightarrow e$ is initiated in every prefix of β $\qquad\qquad$ (3)

(8) α is a prefix of β

$f \in \alpha \Rightarrow e$'s initiating entry precedes f in α, hence in β [(9)+(5)]. f is in

β but not in α and $n > 0 \Rightarrow$ the source in $T(f)$ is Source(b,S,Ω) for some

arc $b \Rightarrow c \in St-DL$ and there are exactly n firings of c in Ω [(7)+Def. 4.2-5]

$\Rightarrow$ there are $In(I(d))$ input entries to e in $\omega(S,\Omega) = \alpha$ [Lemma 4.3-1] $\Rightarrow$

e's initiating entry is in α, hence it precedes f in β [(8)+Def. 4.2-6].

Therefore,

(9) β is causal $\qquad\qquad\qquad\qquad\qquad$ (5)+(6)

(10) The destinations in the transfers of the entries in α are all

distinct, and each of them contains an execution $Ex(d,k)$ where

$d \in St-DL$ and $k > 0$ $\qquad\qquad\qquad\qquad$ (5)

(11) β is α followed by one entry for each arc b holding a token in $S \cdot \Omega$.

The destination in the transfer of each such entry is

$Dst(Ex(c,0),1)$, where c is given by

$\qquad$ if b is the number-i program output arc of P, then $c = (OD,i)$

$\qquad$ otherwise, b is the number-j input arc of an actor labelled d,

$\qquad\quad$ and $c = (d,j)$ $\qquad\qquad\qquad\qquad$ (7)

(12) c is in DL and OD is not the label of an actor in P

Each output arc of P has a unique index [Def. 2.1-1]. Thus the composite

labels c in the target executions of all entries in $\beta-\alpha$ are distinct, and

so the destinations in the transfers of all those entries are distinct

from one another. [(11)+(12)+(4)]. Therefore.

(13) The destinations in the transfers of the entries in β are all
distinct   (10)+(12)

(14) Let $e = Ex(d,k)$ be any execution of which there is an input entry
or an output entry in β. Then either d is the label of an actor
in P or $d \in DL$, so $d \in St$ Def. 4.2-5

(15) $d \in St-DL \Rightarrow$ there are at most $In(l(d))$ input entries to e in β

Lemma 4.3-1

(16) $d \in \{"ID","IT","IF"\} \Rightarrow$ there are 0 input entries to e in $\beta \Rightarrow$ there
are exactly $In(l(d))$ input entries to e in β

(17) Otherwise, $l(d) = OA$ and, for $j \neq 1$, there are no entries whose
transfers have destination $Dst(e,j)$ [(11)], and there is at most
one entry with destination $Dst(e,j)$ [(13)], so there are at most
$In(l(d))$ input entries to e in β [Def. 4.2-5].

(18) $\forall f \in \beta$, $T(f)$ has source $Src(Ex(ID,0),1) \Rightarrow V(f)$ is the value of the
token on the number-i program input arc of P in S, and $\forall f \in \beta$, $T(f)$
has source $Src(Ex(IT,0),1) \Rightarrow V(f) = \underline{true}$ and $T(f)$ has source
$Src(Ex(IF,0),1) \Rightarrow V(f) = \underline{false}$, and $\forall f \in \beta$, there is no k and i such
that $T(f)$ has source $Src(Ex(d,k),1)$ for any $d \in DL-\{"ID","IT","IF"\}$.

For any prefix $\Delta\varphi$ of Ω and for any i, tokens appear in the number-i
group of output arcs of actor d in the transition from $S \cdot \Delta$ to $S \cdot \Delta\varphi \Rightarrow$
either φ is a firing of d or d is a Select which is in a pool in $S \cdot \Delta$ but
not in a pool in $S \cdot \Delta\varphi$ [Def. 3.3-9]. For any two distinct prefixes $\Delta_1\varphi_1$
and $\Delta_2\varphi_2$ of Ω, $|\Delta_1\varphi_1| < |\Delta_2\varphi_2|$, and for any Select d, d is in a pool in
both $S \cdot \Delta_1$ and $S \cdot \Delta_2$ but not in a pool in $S \cdot \Delta_1\varphi_1$ or in $S \cdot \Delta_2\varphi_2 \Rightarrow$ there is a
prefix $\Xi\varphi'$ of Ω with $|\Delta_1\varphi_1| < |\Xi\varphi'| \leq |\Delta_2|$ such that d is not in a pool
in $S \cdot \Xi$ but is in a pool in $S \cdot \Xi\varphi' \Rightarrow \varphi'$ is a firing of d $\Rightarrow \Delta_1\varphi_1$ and $\Delta_2\varphi_2$

do not contain the same number of firings of d [Def. 3.3-9]. Therefore,

(19) For any two distinct prefixes $\Delta_1\varphi_1$ and $\Delta_2\varphi_2$ of Ω, any actor d, and

any i, tokens appear in the number-i group of output arcs of d in

both the transitions from $S\cdot\Delta_1$ to $S\cdot\Delta_1\varphi_1$ and from $S\cdot\Delta_2$ to $S\cdot\Delta_2\varphi_2$ ⇒

$\Delta_1\varphi_1$ and $\Delta_2\varphi_2$ do not contain the same number of firings of d

Given a d∈St-DL, k > 0, and i > 0, for every entry in the set {f| T(f)

has source Src(Ex(d,k),i)}, there is a prefix $\Delta\varphi$ of Ω containing exactly

k firings of d such that a token of value V(f) appears on an arc in the

number-i group of output arcs of d in the transition from $S\cdot\Delta$ to $S\cdot\Delta\varphi$

[Lemma 4.3-1]. There is only one such prefix $\Delta\varphi$ of Ω containing exactly

k firings of d [(19)], and all arcs in the number-i group of output arcs

of d get tokens of the same value in any single state transition [Defs.

2.1-5+3.3-9]. Therefore,

(20) All entries in the set {f| T(f) has source Src(Ex(d,k),i)} have the

same value

(21) All entries in β whose transfers have a common source have the

same value   (14)+(18)+(20)

Hence, $\beta = \eta(S,\Omega)$ is a causal computation for Int(P) [(9)+(13)-(17)+(21)+

Def. 4.2-6].

$\triangle$

Appendix D

Proofs from Chapter 5

<u>Lemma 5.2-6</u> Let α and β be any two causal computations for the same
interpretation Int = (St, I, IE) such that either α is a prefix of β or β
is SOE-inclusive of α, and

(1) for any pointer p, p is the value of the output entries in β of a
Copy execution C $\Rightarrow$ the first entry in β with value p is an output
entry of C.

Let e be any structure operation execution initiated in α wrt Int. Then
for any Assign, Update, or Delete execution A, e is in R(A) in β iff
e is in R(A) in α only if A is initiated in α.

<u>Proof</u>:

Key definitions: Def. 4.2-6 - initiated; Def. 5.1-4 - access history;
Defs. 5.1-5+5.1-7 - durations; Defs. 5.1-6+5.1-8 - reaches;
Def. 5.2-8 - SOE-inclusive

Proof is by induction on the number of structure operation executions
initiated in any prefix of α. Induction hypothesis is that the Lemma is
true of each such execution e initiated in a prefix of α containing the
initiating entries to n such executions. (All initiations are wrt Int.)

Basis: n = 0. Vacuously true.

Induction step: Assume that the Lemma is true for any prefix of α in
which there are n structure operation executions initiated, and consider
prefix γ in which there are n+1.

(2) Let e be any structure operation execution initiated in α, and let e' be any other structure operation execution. Let the label in e be d and the label in e' be d'. There there are $In(l(d))$ input entries to e in α.

(3) α is a prefix of $\beta \Rightarrow e'$ is initiated before e in β iff there is a prefix δ of β containing $In(l(d'))$ input entries to e' but fewer than $In(l(d))$ input entries to e iff there is a prefix δ of α containing $In(l(d'))$ input entries to e' but fewer than $In(l(d))$ input entries to e iff e' is initiated before e in α (2)

(4) e' is initiated before e in β iff e' is initiated before e in α (by definition if β is SOE-inclusive of α)   (3)+(2)

(5) α is a prefix of $\beta \Rightarrow$ all input entries to e in β are in α (2)

(6) For any structure operation execution e initiated in α and any integer j, $V(Ent_\alpha(e,j)) = V(Ent_\beta(e,j))$ (2)+(5)+Def. 5.2-8

(7) Let e and e' be any two distinct structure operation executions such that e is initiated in α. For any pointer p, $Ent_\beta(e',1)$ precedes $Ent_\beta(e,1)$ in H_p^β iff e' initiates before e in β and $V(Ent_\beta(e',1)) = V(Ent_\beta(e,1)) = p$ iff e' is initiated before e in α [(2)+(4)] and $V(Ent_\alpha(e,1)) = V(Ent_\beta(e,1)) = p = V(Ent_\beta(e',1)) = V(Ent_\alpha(e',1))$ [(6)] iff $Ent_\alpha(e,1)$ precedes $Ent_\alpha(e',1)$ in H_p^α

(8) Assume that either $Ent_\beta(e',1)$ does precede $Ent_\beta(e,1)$ in H_p^β or $Ent_\alpha(e',1)$ does precede $Ent_\alpha(e,1)$ in H_p^α. Then e' is initiated before e in either α or β

(9) e' is initiated in both α and β (8)+(2)+(4)

(10) e' is in APS in α iff e' is in APS in β (9)+Def. 5.1-5

(11) For every Update or Delete execution U initiated in α, e' is also

an Update or Delete execution $\Rightarrow V(Ent_\alpha(e',2)) = V(Ent_\alpha(U,2))$ iff $V(Ent_\beta(e',2)) = V(Ent_\beta(U,2))$ $[(9)+(6)]$

(12) For any Update or Delete execution U initiated in α, e' is in SPS(U) iff e' is an Update or Delete execution initiated in α and $V(Ent_\alpha(e',2)) = V(Ent_\alpha(U,2))$ [Def. 5.1-7] iff e' is an Update or Delete execution initiated in β and $V(Ent_\beta(e',2)) = V(Ent_\beta(U,2))$ $[(9)+(11)]$ iff e' is in SPS(U) in β

(13) Let $e = Ex(d,k)$ be any structure operation execution initiated in γ. Then there are $In(I(d))$ input entries to e in γ, hence in α, so e is initiated in α

(14) For any pointer p, $Ent_\beta(e,1)$ is in H_p^β and p is the value of the output entries in β of a Copy execution $C \Rightarrow V(Ent_\beta(e,1)) = p$

(15) $\wedge$ the first entry in β with value p is an output entry of C (1)
$\Rightarrow$ by causality, the initiating entry of C strictly precedes the first entry in β with value p [Def. 4.2-7] $\Rightarrow$ the initiating entry to C strictly precedes $Ent_\beta(e,1)$ $[(14)]$; i.e., C is initiated before e in β

(16) $\Rightarrow$ C is initiated before e in α (13)+(8)
$\Rightarrow$ C is initiated in a prefix of α in which there are fewer than n+1 structure operation executions initiated $[(13)]$ $\Rightarrow$

(17) for any Assign, Update, or Delete execution A, C is in R(A) in β iff C is in R(A) in α only if A is initiated in α [ind. hyp.] $\Rightarrow$ $Ent_\beta(C,1)$ is in duration D(A) in β iff $Ent_\alpha(C,1)$ is in D(A) in α only if A is initiated in α

(18) $Ent_\beta(e,1)$ is in H_p^β and p is the value of the output entries in β of a Copy execution $C \Rightarrow [\alpha$ is a prefix of $\beta \Rightarrow Ent_\beta(e,1)$ is in α $[(13)+(2)+(5)]$ $\Rightarrow$ there is an output entry of C in α with value p

$[(14)+(15)]]$ and $[\beta$ is SOE-inclusive of $\alpha \Rightarrow C$ has output entries

in α $[(14)+(16)] \Rightarrow$ those entries in α have the same value p as

the output entries of C in β [Def. 4.2-5]] $\Rightarrow$ p is the value in α

of the output entries of a Copy execution C [Def. 4.2-6]

(19) e is initiated in β (13)+Defs. 4.2-6+5.2-8

(20) For any pointer p, $Ent_\alpha(e,1)$ is in H_p^α and p is the value of the

output entries in α of a Copy execution C $\Rightarrow V(Ent_\alpha(e,1)) = p \land$

by causality, C is initiated in α [Def. 4.2-7] $\land$ there is an entry

f in α such that $T(f)$ has source $Src(C,1)$ or $Src(C,2)$ and $V(f) = p$

[Def. 4.2-5] $\Rightarrow Ent_\beta(e,1)$ has value p $[(13)+(6)] \land$ there is an entry

g in β such that $T(g)$ has source $Src(C,1)$ or $Src(C,2)$ and $V(g) = p$,

whether α is a prefix of β or β is SOE-inclusive of $\alpha \Rightarrow Ent_\beta(e,1)$

is in H_p^β $[(19)] \land$ p is the value of the output entries in β of a

Copy execution C [Def. 4.2-5] $\Rightarrow$ for any Assign, Update, or Delete

execution U, $Ent_\beta(C,1)$ is in $D(U)$ in β iff $Ent_\alpha(C,1)$ is in $D(U)$ in

α only if A is initiated in $\alpha'[(14)+(17)]$

(21) For any Assign, Update, or Delete execution A, $Ent_\alpha(e,1)$ is in $D(A)$

in α iff for some pointer p, either

(21a) $Ent_\alpha(A,1)$ precedes $Ent_\alpha(e,1)$ in H_p^α and every entry which precedes

$Ent_\alpha(e,1)$ but does not precede $Ent_\alpha(A,1)$ in H_p^α is not in APS (or

$(SPS(A))$, or

(21b) every entry which precedes $Ent_\alpha(e,1)$ in H_p^α is not in APS (or $SPS(A)$),

p is the value of the output entries of a Copy execution C in α,

and $Ent_\alpha(C,1)$ is in $D(A)$ in α

iff either $Ent_\beta(A,1)$ precedes $Ent_\beta(e,1)$ in H_p^β and every entry which

precedes $Ent_\beta(e,1)$ but does not precede $Ent_\beta(A,1)$ in H_p^β is not in

APS (or SPS(A)) [(7)+(8)+(10)+(12)] or every entry which precedes $Ent_\beta(e,1)$ in H_p^β is not in APS (or SPS(A)), p is the value of the output entries in β of a Copy execution C, and $Ent_\beta(C,1)$ is is D(A) in β [(8)+(10)+(12)+(20)+(14)+(17)+(18)] iff $Ent_\beta(e,1)$ is in D(A) in β

$Ent_\beta(e,1) \in D(A)$ in $\beta \Rightarrow Ent_\alpha(e,1) \in D(A)$ in $\alpha \Rightarrow$ (21a) or (21b) [(21)].

(21a) $\Rightarrow$ A is initiated in α. (21b) $\Rightarrow$ C is in R(A) in $\alpha \Rightarrow$ A is initiated in α [(21b)+(20)]. Therefore

(22) $Ent_\beta(e,1) \in D(A)$ in $\beta \Rightarrow Ent_\alpha(e,1) \in D(A)$ in $\alpha \Rightarrow$ A is initiated in α

(23) $\Rightarrow V(Ent_\alpha(A,2)) = V(Ent_\beta(A,2)) \land V(Ent_\alpha(e,2)) = V(Ent_\beta(e,2))$ (13)+(6)

(24) For any Assign, Update, or Delete execution A, e is in R(A) in β

iff e and A are executions of one of a few prescribed combinations of operations, $Ent_\beta(e,1)$ is in D(A) in β, and [A is an Update or Delete and e is a Select, Update, or Delete $\Rightarrow V(Ent_\beta(e,2)) = V(Ent_\beta(A,2))$] iff e and A are executions of one of a few prescribed combinations of operations, $Ent_\alpha(e,1)$ is in D(A) in α [(21)] and [A is an Update or Delete and e is a Select, Update, or Delete $\Rightarrow V(Ent_\alpha(e,2)) = V(Ent_\alpha(A,2))$] [(22)+(23)] iff e is in R(A) in α

e is in R(A) in $\alpha \Rightarrow Ent_\alpha(e,1) \in D(A)$ in $\alpha \Rightarrow$ A is initiated in α (24)+(22)

$\triangle$

Lemma 5.2-7 Let $S = (\Gamma, U)$ be any initial standard state for an L_{BS} program P, and let $\theta\varphi$ be any firing sequence starting in S (φ is the last firing). Let $\alpha = \eta(S,\theta)$ and $\beta = \eta(S,\theta\varphi)$. Let f be any entry in β but not in α whose value is some pointer p. If $f = Ent_\beta(e,1)$ for some execution e, then for any other execution e', f is in duration D(e') in β iff D(e') extends to the end of H_p^α. Furthermore, $\theta = \lambda \Rightarrow$ no durations extend to

the end of H_p^α for any pointer p.

Proof:

Key definitions: Def. 5.1-4 - access history; Defs. 5.1-6+5.1-8 - reach;

Defs. 5.1-5+5.1-7 - duration; Def. 5.2-5 - CC

Proof is by induction on the length of θ. Let Int(P) be (St, I, IE).

Basis: $|\theta| = 0$.

(1) $\alpha = \lambda$ and $\beta = \eta(S, \theta)$ consists of input entries to a single execution

Alg. 4.3-1

(2) If there is no pointer-valued entry $Ent_\beta(e,1)$ in $\beta-\alpha$, then the

Lemma is vacuously true. Assume therefore that there is an entry

$f = Ent_\beta(e,1)$ for some execution e, and that V(f) is pointer p

(3) For any execution $e' = Ex(d,k)$, $f \in D(e')$ in $\beta \Rightarrow I(d)$ is Assign,

Update, or Delete and either

(3a) $f = Ent_\beta(e,1)$ and $Ent_\beta(e',1)$ are distinct entries in H_p^β, or

(3b) p is the value of the output entries in β of a Copy execution C   (2)

(4) $\Rightarrow In(I(d)) > 0$ Def. 4.3-1

(3a) $\Rightarrow e' \neq e$ and e' is initiated in $\beta \Rightarrow$ there is an input entry to e'

in β [(4)+Def. 4.2-6]. (3b) $\Rightarrow Ent_\beta(C,1)$ strictly precedes

$f = Ent_\beta(e,1)$ in β [(2)+Lemma 5.2-3]; i.e., there is an input

entry to C$\neq$e in β. Therefore, $f \in D(e') \Rightarrow$ there are input entries

to two distinct executions in β [(3)], so by (1),

(5) For any execution e', f is not in D(e') in β

(6) α is a computation for Int(P)   Lemma 4.3-2

(7) For any execution $e' = Ex(d,k)$, and any pointer p, D(e') extends

to the end of $H_p^\alpha \Rightarrow I(d)$ is Assign, Update, or Delete, and either

(7a) $\text{Ent}_\alpha(e',1)$ is in H_p^α, or (7b) there is a Copy execution C such that

C is in the reach $R(e')$ in α [Def. 5.2-6]. (7a) $\Rightarrow$ e' is initiated in α

wrt $\text{Int}(P)$ [(6)]. (7b) $\Rightarrow \text{Ent}_\alpha(C,1)$ is in duration $D(e')$ in $\alpha \Rightarrow \text{Ent}_\alpha(C,1)$

is in an access history in $\alpha \Rightarrow$ C is initiated in α. Hence, for any

execution e', $D(e')$ extends to the end of $H_p^\alpha \Rightarrow$ there is an execution of

an operation having non-zero input arity initiated in α [Def. 4.3-1] $\Rightarrow$

there is an entry in α [Def. 4.2-6]. By this and (5),

For any execution e', $f \notin D(e')$ and $D(e')$ does not extend to the end of H_p^α.

Induction step: Assume that the Lemma is true for any $\theta\varphi$ in which

$0 \leq |\theta| \leq n$, and consider $\theta\varphi$ in which $|\theta| = n+1$. Let the final firing φ

be the kth firing of the actor labelled d.

(8) β is α followed by input entries to $\text{Ex}(d,k)$, followed possibly by

more entries   Alg. 4.3-1

(9) Assume that there is an entry f in $\beta-\alpha$ whose value is pointer p,

and that $f = \text{Ent}_\beta(e,1)$ for some execution e. Then $e = \text{Ex}(d,k)$ (8)

(10) H_p^α is a prefix of H_p^β, f is in H_p^β, and for every entry $\text{Ent}(e',j)$

in $H_p^\beta - H_p^\alpha$, $e' \neq e \Rightarrow e'$ is not a structure operation execution

                                      (8)+Lemma 5.2-5

(11) Let NAR (NAR') be the node activation record derived from θ and α

($\theta\varphi$ and β). Then ran NAR is consistent with U   Lemma 5.2-2

(12) Let CC_α (CC_β) be the Creating-Copy function corresponding to NAR

(NAR'). $CC_\alpha(p)$ is defined $\Rightarrow \text{NAR}(CC_\alpha(p)) = (p,n)$ for some n

(13) $\Rightarrow CC_\alpha(p)$ is initiated in α Def. 5.2-4

$\wedge$ $(p,n) \in$ ran NAR [Def. 5.2-1] $\Rightarrow \text{NAR}'(CC_\beta(p)) = \text{NAR}(CC_\alpha(p)) = (p,n)$ [(12)+

Lemma 5.2-5] $\wedge$ $p \notin \text{dom } \Pi$ in U [(11)+Def. 5.2-3] $\Rightarrow$

(14) $CC_\beta(p) = CC_\alpha(p)$

(15) ∧ the first entry with value p in β (if any) is an output entry

of $CC_\beta(p)$ and is strictly preceded by $Ent_\beta(CC_\beta(p),1)$

$$(11)+(12)+\text{Lemma } 5.2-3$$

(16) In summary, $CC_\alpha(p)$ is defined ⇒ p is the value in β of the output

entries of a Copy execution $CC_\alpha(p)$, which is initiated in α

(17) There is a Copy execution C whose output entries in β have value p

⇒ $CC_\beta(p)$ is defined, C = $CC_\beta(p)$, and $Ent_\beta(C,1)$ strictly precedes

the first entry in β with value p, which is an output entry of C

$$(11)+(12)+\text{Lemma } 5.2-3$$

α is a prefix of β, and both are causal computations for Int(P) [(8)+

Lemma 4.3-2], so

(18) For any Copy execution C initiated in α and any other execution e',

C is in the reach R(e') in α iff C is in R(e') in β (17)+Lemma 5.2-6

(19) For any execution e', $CC_\alpha(p)$ is defined and $CC_\alpha(p)$ is in R(e') in

α ⇒ p is the value of the output entries in β of a Copy execution

C, and C is in R(e') in β [(16)+(18)] ⇒ p is the value in β of

the output entries of a Copy execution C, and $Ent_\beta(C,1) \in D(e')$ in β

(20) There is a Copy execution C whose output entries in β have value p

⇒ $Ent_\beta(C,1)$ strictly precedes $Ent_\beta(e,1)$, so C≠e   (17)+(9)

(21) ∧ NAR'(C) = (p,n) for some node n   (17)+(12)

(22) ⇒ $Ent_\beta(C,1)$ is in α ⇒ C is initiated in α   (8)+Def. 4.2-6

⇒ NAR(C) = NAR'(C) = (p,n) for some n [(21)+Lemma 5.2-5]

(23) ⇒ $CC_\alpha(p)$ is defined and equals C

(24) For any execution e', there is a Copy execution C whose output

entries in β have value p and $Ent_\beta(C,1) \in D(e')$ in β ⇒ there is a

Copy execution C whose output entries in β have value p and

$C \in R(e')$ in $\beta \Rightarrow CC_\alpha(p)$ is defined and $C \in R(e')$ in α

$$(20)+(23)+(22)+(18)$$

$Ent_\beta(e',1)$ precedes $f = Ent_\beta(e,1)$ in H_p^β and e' is a structure operation execution $\Rightarrow Ent_\beta(e',1)$ is in H_p^α [(10)], so

(25) $e' = Ex(c,n)$ is a structure operation execution and either $Ent_\beta(e',1)$
    precedes f in H_p^β or $Ent_\alpha(e',1)$ is in $H_p^\alpha \Rightarrow e'$ is initiated in $\alpha \wedge$
    $In(I(c)) > 0$ [Defs. 4.3-2+4.3-1+2.1-5] $\Rightarrow Ent_\beta(e',1) = Ent_\alpha(e',1)$

$$(8)+Def. 4.2-6$$

For any Assign execution e', $D(e')$ extends to the end of H_p^α iff

    a. $Ent_\alpha(e',1)$ is the last input entry to an Assign execution in H_p^α

or b.   there is no input entry to an Assign execution in H_p^α, $CC_\alpha(p)$ is
    defined, and $CC_\alpha(p) \in R(e')$ in α [Def. 5.2-6] iff

    a. $Ent_\beta(e',1)$ is the last input entry to an Assign execution
    preceding f in H_p^β [(10)+(25)],

or b.   there is no input entry to an Assign execution preceding f in
    H_p^β, there is a Copy execution C whose output entries in β have
    value p, and $Ent_\beta(C,1) \in D(e')$ in β [(10)+(19)+(24)] .

iff $f \in D(e')$ in β.

Replacing "input entry to an Assign execution" with "number-1 input entry to an Update/Delete execution having a particular selector input" in the above paragraph yields a proof that:

For any Update/Delete execution e', $D(e')$ extends to the end of H_p^α iff $f \in D(e')$ in β.

$\triangle$

Theorem 5.2-1 Let $S = (\Gamma, U)$ be any initial standard state for an L_{BS} program P, and let Ω be any firing sequence starting in S. Let $\omega = \eta(S, \Omega)$ and let NAR be the node activation record derived from Ω and ω. Then the heap determined by ω from U and NAR is defined and is identical to the heap in the state $S \cdot \Omega$.

Proof:

Key definitions: Def. 2.2-5 - structure operations; Alg. 4.3-1 - $\eta(S, \Omega)$; Def. 5.2-1 - node activation record; Def. 5.2-5 - Creating Copy function; Def. 5.2-6 - durations extending to the end of an access history; Def. 5.2-7 - heap determined by ω from U and NAR

Since ω is a computation for Int(P) [Lemma 4.3-2] and NAR is compatible with ω and ran NAR is consistent with U [Lemma 5.2-2], the heap determined by ω from U and NAR is defined [Def. 5.2-7]. Prove the rest of the theorem by induction on the length of Ω. Let $U = (N_0, \Pi_0, SM_0)$.

Basis: $|\Omega| = 0$. Let (N, Π, SM) be the heap determined by $\omega = \eta(S, \Omega)$.

(1) $S \cdot \Omega = S$, so the heap in $S \cdot \Omega$ is (N_0, Π_0, SM_0) Def. 2.3-1

There are no entries in ω. Since In(Copy) = 1, there are no Copy executions initiated in ω [Defs 5.1-1+4.2-6]. Since ran NAR is empty, $\Pi = \Pi_0$ and $N = N_0$ [Def. 2.2-1]. Let (p, n) be any pair in Π. Then no durations extend to the end of H_P^{ω} [Lemma 5.2-7]. $SM(n) = SM_0(m)$ where, since $(p, n) \in \Pi_0$, $m = n$. Therefore, (N, Π, SM) is the heap in $S \cdot \Omega$ [(1)].

Induction step: Assume that the Theorem is true for any Ω of length $n \geq 0$. Consider firing sequence $\theta\varphi$ of length $n+1$, in which the last firing φ is the kth firing of the actor in P labelled d. Let $\alpha = \eta(S, \theta)$ and $\beta = \eta(S, \theta\varphi)$, and let (N, Π, SM) and (N', Π', SM') be the heaps in $S \cdot \theta$

and $S \cdot \theta \varphi$ respectively.

(2) β is α followed by m input entries to an execution e = Ex(d,k), where m is the number of tokens removed by φ, followed possibly by input entries to executions Ex(c,i) where c is in DL   Def. 4.3-1

(3) Any execution Ex(c,n)$\neq$e initiated in β but not in α has input entries in $\beta - \alpha$, and so is not a structure operation execution

$$(2) + Defs. \ 4.2-6 + 4.3-2$$

(4) For any pointer p, H_p^α is a prefix of H_p^β, any input entries to e which have value p are in H_p^β, and for any entry Ent(e',j) in $H_p^\beta - H_p^\alpha$, e'$\neq$e $\Rightarrow$ e is not a structure operation execution   Lemma 5.2-5

Consider first the consequences of φ's **not** being a firing of certain types of actors. Let NAR (NAR') be the node activation record derived from θ and α ($\theta\varphi$ and β) given Int(P). Let $(N_\alpha, \Pi_\alpha, SM_\alpha)$ and $(N_\beta, \Pi_\beta, SM_\beta)$ be the heaps determined by α from U and NAR and by β from U and NAR'.

(5) $\Pi_\alpha = \Pi$ and $N_\alpha = N$ ind. hyp.

(6) φ is not a Copy firing $\Rightarrow$ N' = N and Π' = Π [Def. 2.3-1] $\wedge$ the set C of Copy executions initiated in α equals the set of Copy executions initiated in β [(3)] $\Rightarrow$ for any C$\in$C, NAR'(C) = NAR(C) $\Rightarrow$ ran NAR' = ran NAR [Lemma 5.2-5 + Def. 5.2-4] $\Rightarrow$ N' = N = N_α = N_β and Π' = Π = Π_α = Π_β [(5)]

(7) NAR' is a node activation record and ran NAR' is consistent with U

Lemma 5.2-2

Hence, for any pointer p, there is at most one Copy execution C such that NAR'(C) has p in it [Def. 5.2-3].

(8) For any n, (p,n)$\in \Pi - \Pi_0$ $\Rightarrow$ $\exists$Copy execution C: NAR(C) = (p,n) and C is initiated in α [(5)] $\Rightarrow$ NAR'(C) = (p,n) [Lemma 5.2-5]

(9) Let CC_α (CC_β) be the Creating-Copy function corresponding to NAR (NAR'). $CC_\beta(p)$ is defined $\Rightarrow$ there is an n such that NAR'($CC_\beta(p)$) is defined and equal to (p,n) $\Rightarrow$ (p,n)$\notin\Pi_0$ [(7)+Def. 5.2-3]

(10) $CC_\alpha(p)$ is defined $\Rightarrow$ there is a Copy execution C and a pointer n such that NAR(C) = (p,n) $\Rightarrow$ C is initiated in α [Def. 5.2-4] $\Rightarrow$ NAR'(C) = (p,n) [Lemma 5.2-5] $\Rightarrow$ $CC_\beta(p)$ is defined

(11) (p,n)$\in\Pi$ and $CC_\beta(p)$ is defined $\Rightarrow$ (p,n)$\in\Pi-\Pi_0$ [(9)]

$\Rightarrow$ $\exists$Copy execution C initiated in α and NAR(C) = NAR'(C) = (p,n) [(8)] $\Rightarrow$

(12) $Ent_\beta(C,1)$ is in α [(2)+Def. 4.2-6]

(13) $\wedge$ $CC_\alpha(p)$ is defined and equal to C = $CC_\beta(p)$

(14) $\Rightarrow$ $Ent_\alpha(CC_\alpha(p),1)$ = $Ent_\beta(CC_\beta(p),1)$ [(12)]

Both α and β are causal computations for Int(P) [Lemma 4.3-2] and α is a prefix of β [(2)]. For any pointer p, p is the value of the output entries in β of a Copy execution C $\Rightarrow$ the first entry in β with value p is an output entry of C [Lemma 5.2-3]. From these

(15) For any (p,n)$\in\Pi$ such that $CC_\beta(p)$ is defined, and for any Assign, Update, or Delete execution A, $CC_\beta(p)$ is in reach R(A) in β iff $CC_\alpha(p)$ is in R(A) in α (11)+(14)+Lemma 5.2-6

Consider now the case that for some (p,n)$\in\Pi$, φ is not an Assign firing which removes a token of value p.

(16) SM'(n) has the same value as SM(n)

Also, e is not an Assign execution with $V(Ent_\beta(e,1))$ = p [(2)], so e is not an Assign execution with $Ent_\beta(e,1)$ in H_p^β. Therefore, from (4),

(17) H_p^α is a prefix of H_p^β and there are no Assign execution input entries in $H_p^\beta-H_p^\alpha$

(18) For any Assign execution A, D(A) extends to the end of $H_p^\beta \Rightarrow$ either $\text{Ent}_\beta(A,1)$ is the last input entry to an Assign execution in H_p^β, or there is no such entry, and $CC_\beta(p)$ is defined and is in R(A) in β $\Rightarrow$ either $A \neq e$ and $\text{Ent}_\alpha(A,1)$ is the last input entry to an Assign execution in H_p^α [(17)], or there is no such entry in H_p^α [(17)] and $CC_\alpha(p)$ is defined and is in R(A) in α [(11)+(13)+(15)]

(19) For any Assign execution A, D(A) extends to the end of $H_p^\beta \Rightarrow A \neq e$ and D(A) extends to the end of H_p^α [(18)], and the value in $SM_\beta(n)$ is $V(\text{Ent}_\beta(A,2)) \Rightarrow$ the value in $SM_\alpha(n)$ is $V(\text{Ent}_\alpha(A,2)) \Rightarrow$ the values in $SM_\beta(n)$ and $SM_\alpha(n)$ are the same [(2)+(3)]

(20) $(p,n) \in \Pi - \Pi_0 \Rightarrow \text{Ent}_\alpha(CC_\alpha(p),1) = \text{Ent}_\beta(CC_\beta(p),1)$, and that entry is in α [(11)+(14)+(12)] $\Rightarrow V(\text{Ent}_\beta(CC_\beta(p),1))$ is dynamically descended from a pointer q in β only if $V(\text{Ent}_\alpha(CC_\alpha(p),1))$ is dynamically descended from q in α [(2)+(3)+Def. 5.1-9]

(21) There is no Assign execution A such that D(A) extends to the end of $H_p^\beta \Rightarrow$ there is no Assign input entry in H_p^β and, even if $CC_\beta(p)$ is defined, there is no Assign execution A such that $CC_\beta(p)$ is in R(A) in $\beta \Rightarrow$ there is no Assign input entry in H_p^α [(17)] and even if $CC_\alpha(p)$ is defined, there is no Assign execution A such that $CC_\alpha(p)$ is in R(A) in α [(10)+(15)] $\Rightarrow$ there is no Assign execution A such that D(A) extends to the end of $H_p^\alpha \Rightarrow$ if $(p,n) \in \Pi_0$, the value in $SM_\beta(n)$ and the value in $SM_\alpha(n)$ are both equal to the value in $SM_0(n)$, and if $(p,n) \in \Pi - \Pi_0$, the value in $SM_\beta(n)$ is the value in $SM_0(m)$, where (q,m) is that unique pair in Π_0 such that $V(\text{Ent}_\beta(CC_\beta(p),1))$ is dynamically descended from q in β, in which case, the value in $SM_\alpha(n)$ is also the value in $SM_0(m)$ [(20)]

I.e., there is no Assign execution A such that D(A) extends to the

end of H_p^β $\Rightarrow$ the value in $SM_\beta(n)$ equals the value in $SM_\alpha(n)$

(22) For all $(p,n) \in \Pi$, φ is not an Assign firing which removes a token

of value p $\Rightarrow$ the value in $SM'(n)$ equals the value in $SM_\beta(n)$

$$(16)+(19)+(21)+\text{ind. hyp.}$$

Replacing "Assign firing (execution)" with "Update/Delete firing

(execution)" in (16)-(21) yields a proof of

(23) For any $(p,n) \in \Pi$ and selector $s \in \Sigma$, φ is not an Update/Delete firing

with pointer input p and selector input s $\Rightarrow$ there is a pair (s,r),

for some node r, in $SM_\beta(n)$ iff (s,r) is in $SM'(n)$.

There are now four cases to consider, based on the type of actor of which

φ __is__ a firing.

Case I: φ is not a firing of a Copy, Assign, Update, or Delete.

$N' = N_\beta$, $\Pi' = \Pi_\beta$, and $\Pi' = \Pi$ [(6)]. For all $(p,n) \in \Pi'$,

$SM_\beta(n) = SM'(n)$ [(22)+(23)]. I.e., $(N_\beta, \Pi_\beta, SM_\beta) = (N', \Pi', SM')$

Case II: φ is an Assign firing.

(24) $N_\beta = N'$, $\Pi_\beta = \Pi'$, and $N' = N$ (6)

(25) Let p be the pointer input to φ, and let $n = \Pi(p)$. Then for all

$m \neq n \in N'$, $SM_\beta(m) = SM'(m)$, and $SM_\beta(n)$ has the same ordered pairs as

$SM'(n)$ (24)+(22)+(23)

The value in $SM'(n)$ is equal to v, the value of the token removed from

d's number-2 input arc by φ [(25)]. e is an Assign execution, $V(\text{Ent}_\beta(e,1))$

is p, and $V(\text{Ent}_\beta(A,2)) = v$ [(2)+(23)]. $\text{Ent}_\beta(e,1)$ is the last input entry

to an Assign execution in H_p^β [(4)], so duration $D(e)$ extends to the end

of H_p^β. Therefore, the value in $SM_\beta(n)$ is v. Hence

if φ is an Assign firing, then $(N_\beta, \Pi_\beta, SM_\beta) = (N', \Pi', SM')$.

Case III: φ is an Update or Delete firing

(26) $N_\beta = N'$, $\Pi_\beta = \Pi'$, $N' = N$, and $\Pi' = \Pi$ (6)

(27) Let p be the pointer input to φ, let $n = \Pi(p)$, and let s be the

     selector input to φ. Then for all $m \neq n \in N'$, $SM_\beta(m) = SM'(m)$, $SM_\beta(n)$

     has the same value as $SM'(n)$, and for all $s' \neq s$, there is a pair

     (s', r), for some node r, in $SM_\beta(n)$ iff $(s', r) \in SM'(n)$ (26)+(22)+(23)

(28) e is an execution of the same action of which φ is a firing,

     $V(Ent_\beta(e,1)) = p$, and $V(Ent_\beta(e,2)) = s$ [(2)+(27)]. $Ent_\beta(e,1)$

     is the last input entry to an Update or Delete execution with

     selector input s in H_p^β [(4)], so $D(e)$ extends to the end of H_p^β

(29) φ is a Delete firing $\Rightarrow$ there is no pair in $SM'(n)$ with s in it

     [(27)] and e is a Delete execution, so there is no pair in $SM_\beta(n)$

     with s in it [(28)], from which, $SM_\beta(n) = SM'(n)$ [(27)]

(30) If φ is an Update firing, let q be the pointer it removes from d's

     number-3 input arc. Then the pair $(s, \Pi(q))$ is in $SM'(n)$, and is

     the only pair containing s in $SM'(n)$ [(27)], and e is an Update

     execution with $V(Ent_\beta(e,3)) = q$ [(2)], so the pair $(s, \Pi_\beta(q))$ is

     in $SM_\beta(n)$ and is the only pair containing s in $SM_\beta(n)$ [(28)];

     i.e., the pair $(s, \Pi(q))$ is in $SM_\beta(n)$ [(26)]. Therefore,

     $SM_\beta(n) = SM'(n)$ [(27)]

Hence, if φ is either an Update or a Delete firing,

$(N_\beta, \Pi_\beta, SM_\beta) = (N', \Pi', SM')$ [(26)+(27)+(29)+(30)].

Case IV: φ is a Copy firing.

(31) For all $(p,n) \in \Pi$, $SM_\beta(n) = SM'(n)$ (22)+(23)

(32) Let φ be $(d, (p,n))$. Then $\Pi' = \Pi \cup \{(p,n)\}$ and $N' = N \cup \{n\}$ Def. 2.3-1

Let C be the set of Copy executions initiated in α. Then the set of Copy

executions initiated in β is $C \cup \{e\}$. [(2)+(3)]. For all $C \in C$,

NAR'(C) = NAR(C) [Lemma 5.2-5]. Hence ran NAR' = ran NAR$\cup$NAR'(e). From

this and

(33) NAR'(e) is the ordered pair in the kth firing of d in $\theta \varphi$, which is

the ordered pair in φ, which is (p,n)   (2)+(32)+Def. 5.2-4

it is seen that $\Pi_\beta = \Pi_0 \cup$ran NAR' = $\Pi_0 \cup$ran NAR$\cup\{(p,n)\}$ = $\Pi_\alpha \cup \{(p,n)\}$.

Then, $N_\beta = N_\alpha \cup \{n\}$. Therefore,

(34) $N_\beta = N'$ and $\Pi_\beta = \Pi'$ [(32)+(5)]

(35) Let q be the value of the token removed by φ, and let m = Π(q).

Then SM'(n) = SM(m)   (32)

(36) $CC_\beta(p)$ is defined and equals e   (33)

$Ent_\beta(e,1)$ strictly precedes the first entry, if any, in β with value p

[(36)+Lemma 5.2-3], so there is no entry in α with value p [(2)]. Thus

if there is an entry with value p in β, it is not an input entry to a

structure operation execution [(3)]. Therefore,

(37) H_p^β contains no input entries to structure operation executions   (3)

(38) $Ent_\beta(e,1)$ is in β but not in α and its value is q   (2)+(35)

Therefore,

(39) For all executions e', duration D(e') extends to the end of H_p^β

iff $e \in R(e')$ in β [(37)+(36)] iff $Ent_\beta(e,1) \in D(e')$ in β [Defs. 5.1-6+

5.1-8] iff D(e') extends to the end of H_q^α [(38)+Lemma 5.2-7]

Since $(p,n) \notin \Pi_0$ [(7)+(33)],

(40) Letting r be the unique pointer in dom Π_0 from which

q = $V(Ent_\beta(CC_\beta(p),1))$ is dynamically descended in β, $SM_\beta(n)$ depends

on $SM_0(\Pi_0(r))$ and on the input entries to the executions whose

durations extend to the end of H_p^β (38)+(36)

(41) $(q,m) \in \Pi$ [(35)+Thm. 2.2-1]. $(q,m) \in \Pi_0 \Rightarrow q = r$ [(40)+(35)].

(42) $(q,m) \notin \Pi_0 \Rightarrow (q,m) \in \Pi - \Pi_0$ [(41)] $\Rightarrow r \neq q \wedge CC_\beta(q)$ is defined and q is the

value of its output entries in β [(40)+Lemma 5.2-3] $\wedge$

$Ent_\alpha(CC_\alpha(q),1) = Ent_\beta(CC_\beta(q),1)$, and that entry is in α [(11)+(14)+

(12)] $\Rightarrow V(Ent_\beta(CC_\beta(q),1))$ is dynamically descended in β from every

pointer except q from which q is dynamically descended in β

[(40)+Def. 5.1-9] $\Rightarrow V(Ent_\alpha(CC_\alpha(q),1))$ is dynamically descended in

β from r $\Rightarrow V(Ent_\alpha(CC_\alpha(q),1))$ is dynamically descended in α from

r [(2)+(3)+Def. 5.1-9] $\Rightarrow$ r is the unique pointer in dom Π_0 from

which $V(Ent_\alpha(CC_\alpha(q),1))$ is dynamically descended in α [Lemma 5.2-4]

$SM_\alpha(m)$ depends in the same way on $SM_0(\Pi_0(r))$ and on the input entries

to the executions whose durations extend to the end of H_q^α [(40)+(41)+(42)].

The same executions' durations extend to the end of H_p^β and H_q^α, and their

input entries are the same in α and β [(39)+(19)+(2)+(3)]. Therefore,

$SM_\beta(n) = SM_\alpha(m)$ [(40)]. By induction hypothesis, $SM_\alpha(m) = SM(m)$. So

$SM_\beta(n) = SM'(n)$ [(35)]. Hence, $(N_\beta, \Pi_\beta, SM_\beta) = (N', \Pi', SM')$ [(34)+(31)+

(32)].

                                                    Q.E.D.

Theorem 5.3-1 For any two equal standard states S_1 and S_2 for the same program P, and any two equal firing sequences Ω_1 starting in S_1 and Ω_2 starting in S_2, $S_2 \cdot \Omega_2$ equals $S_1 \cdot \Omega_1$. Furthermore, if I is the mapping under which the conditions of each arc b in P match in S_1 and S_2, then the mapping under which the conditions of b in $S_1 \cdot \Omega_1$ and $S_2 \cdot \Omega_2$ match is

$IU\{(n_1,n_2) \mid \exists k: \text{for } i=1,2, n_1 \text{ is the node in the } k^{th} \text{ firing in } \Omega_1\}.$

Proof:

Key definitions: Def. 2.1-5 – non-structure operations;

Def. 2.2-5 – structure operations; Def. 2.4-1 – equal components;

Def. 2.4-2 – Match

Proof is by induction on the length of Ω_1.

Basis: $|\Omega_1| = 0$. Then $|\Omega_2| = 0$ [Def. 2.4-5], so $S_2 \cdot \Omega_2 = S_2$ and $S_1 \cdot \Omega_1 = S_1$ [Def. 2.3-1]. By hypothesis, then, $S_2 \cdot \Omega_2$ equals $S_1 \cdot \Omega_1$.

Induction step: Assume the Theorem is true if Ω_1 is of length $n \geq 0$, and consider equal firing sequences $\Omega_1 \varphi_1$ and $\Omega_2 \varphi_2$, starting in S_1 and S_2, where $\Omega_1 \varphi_1$ is of length n+1 and its last firing φ_1 is of the actor in P labelled d.

(1) $\Omega_2 \varphi_2$ is also of length n+1 and φ_2 is also a firing of d   Def. 2.4-5

$S_2 \cdot \Omega_2$ equals $S_1 \cdot \Omega_1$ by induction hypothesis, so

(2) There is a one-to-one mapping I such that, for each arc b in P,

Match$((b,S_2 \cdot \Omega_2), I, (b,S_1 \cdot \Omega_1))$ Def. 2.4-3

(3) If d is a gate actor, then the control token removed by φ_1 has the same value as the control token removed by φ_2 (2)

(4) For each arc b in P, b has no token in $S_1 \cdot \Omega_1 \varphi_1$ iff either

b is neither an input nor output arc of d and has no token in $S_1 \cdot \Omega_1$,

b is an input arc of d from which φ_1 removes an input token, or

b is an output arc of T- or F-gate d and φ_1 places no token on b

iff b is neither an input nor output arc of d and has no token in

$S_2 \cdot \Omega_2$ [(2)] or b is an input arc of d from which φ_2 removes a token

or b is an output arc of T- or F-gate d and φ_2 places no token

on b [(3)] iff b has no token in $S_2 \cdot \Omega_2 \varphi_2$

At this point, it will be helpful to introduce the following notation:
For any state S of any program, and for any arc b of the program which
holds a token in S, denote by $TV(b,S)$ the value of that token.

(5) For each arc b in P which has a token in $S_1 \cdot \Omega_1 \varphi_1$ and is not an

output arc of d, $TV(b,S_1 \cdot \Omega_1 \varphi_1) = TV(b,S_1 \cdot \Omega_1)$ and

$$TV(b,S_2 \cdot \Omega_2 \varphi_2) = TV(b,S_2 \cdot \Omega_2) \tag{4}$$

(6) $TV(b,S_1 \cdot \Omega_1)$ is not a pointer $\Rightarrow TV(b,S_2 \cdot \Omega_2) = TV(b,S_1 \cdot \Omega_1)$ [(2)] $\Rightarrow$

$TV(b,S_2 \cdot \Omega_2 \varphi_2) = TV(b,S_1 \cdot \Omega_1 \varphi_1)$ [(5)]

(7) For $i = 1,2$, let the heap in $S_i \cdot \Omega_i$ be $U_i = (N_i, \Pi_i, SM_i)$, and let the

heap in $S_i \cdot \Omega_i \varphi_i$ be $U_i' = (N_i', \Pi_i', SM_i')$. For any arc b, $TV(b,S_1 \cdot \Omega_1)$

is a pointer $p_1 \Rightarrow$ letting $p_2 = TV(b,S_2 \cdot \Omega_2)$, $U_2 . \Pi_2(p_2) \overset{I}{=} U_1 . \Pi_1(p_1)$

[(2)] $\Rightarrow I(\Pi_1(p_1)) = \Pi_2(p_2)$, and, for any node n equal to or reach-

able from $\Pi_1(p_1)$ in U_1, $SM_2(I(n)) = I(SM_1(n))$

(8) Let q_1 and q_2 be the values of the number-1 input tokens to φ_1 and

φ_2 respectively, and for $i=1,2$, let $m_i = \Pi_i(q_i)$. Then $m_2 = I(m_1)$

$$\tag{7}$$

(9) For all $(q,m) \in \Pi_1$, $m \neq m_1$ or φ_1 is not an Assign, Update, or Delete

firing $\Rightarrow SM_1'(m) = SM_1(m)$

(10) For all $(q,m) \in \Pi_1$ such that m equals or is reachable from $\Pi_1(p)$ for

any pointer p on an arc in $S_1 \cdot \Omega_1$, $m \neq m_1$ or φ_1 is not an Assign,

Update, or Delete firing $\Rightarrow m_2 \neq I(m)$ or φ_2 is not an Assign, Update, or Delete firing $[(8)+(2)+(1)]$, and $SM_2(I(m)) = I(SM_1(m))$ $[(7)] \Rightarrow$ $SM_2'(I(m)) = SM_2(I(m)) = I(SM_1(m)) = I(SM_1'(m))$ $[(8)+(9)]$

(11) If $SM_1(m_1) = \{v, (s_1,n_1),\ldots,(s_j,n_j)\}$, then

$$SM_2(m_2) = \{v, (s_1,I(n_1)),\ldots,(s_j,I(n_j))\} \tag{8}$$

(12) Let b be d's number-2 input arc (if any). φ_1 is an Assign firing $\Rightarrow$ $SM_1'(m_1) = \{v', (s_1,n_1),\ldots,(s_j,n_j)\}$ where $v' = TV(b,S_1 \cdot \Omega_1)$ $[(11)]$ $\Rightarrow \varphi_2$ is an Assign firing with $TV(b,S_2 \cdot \Omega_2) = v'$ $[(1)+(6)] \Rightarrow$ $SM_2'(m_2) = \{v', (s_1,I(n_1)),\ldots,(s_j,I(n_j))\}$ $[(11)] \Rightarrow$ $SM_2'(m_2) = I(SM_1'(m_1))$

(13) φ_1 is a Delete firing $\Rightarrow SM_1'(m_1)$ is $SM_1(m_1)$ minus the pair with selector $s = TV(b,S_1 \cdot \Omega_1)$ (if any) $\Rightarrow \varphi_2$ is a Delete firing and $TV(b,S_2 \cdot \Omega_2) = s$ $[(1)+(6)] \Rightarrow SM_2'(m_2)$ is $SM_2(m_2)$ minus the pair with selector s (if any) $\Rightarrow SM_2'(m_2) = I(SM_1'(m_1))$ $[(11)]$

(14) φ_1 is an Update firing $\Rightarrow SM_1'(m_1) = SM_1(m_1)$ with any pair having s in it replaced by $(s, \Pi_1(q_1'))$, where $s = TV(b,S_1 \cdot \Omega_1)$ and for c d's number-3 input arc, $TV(c,S_1 \cdot \Omega_1) = q_1' \Rightarrow \varphi_2$ is an Update firing, $TV(b,S_2 \cdot \Omega_2) = s$, and $TV(c,S_2 \cdot \Omega_2) = q_2'$ such that $\Pi_2(q_2') = I(\Pi_1(q_1'))$ $[(1)+(6)+(7)] \Rightarrow SM_2'(m_2)$ is $SM_2(m_2)$ with any pair having s in it replaced by $(s,I(\Pi_1(q_1'))) \Rightarrow SM_2'(m_2) = I(SM_1'(m_1))$ $[(11)]$

(15) $\Pi_1 \subseteq \Pi_1'$ and $\Pi_2 \subseteq \Pi_2'$ $\tag{7}$

(16) Let b be any arc which is not an output arc of d such that $TV(b,S_1 \cdot \Omega_1 \varphi_1)$ is some pointer p_1. Then p_1 is on an arc in $S_1 \cdot \Omega_1$, and $TV(b,S_2 \cdot \Omega_2 \varphi_2) = TV(b,S_2 \cdot \Omega_2)$ is a pointer p_2 such that

$$I(\Pi_1(p_1)) = \Pi_2(p_2) \tag{5+7}$$

(17) $I(\Pi_1'(p_1)) = \Pi_2'(p_2)$ $\tag{15+16}$

(18) For any pointer p∈dom Π_1, n is any node reachable from $\Pi_1'(p)$ in U_1'

⇒ there is a chain of nodes $r_1, r_2, \ldots, r_k$ such that $r_1 = \Pi_1'(p)$,

$r_k = n$, and for i=1,...,k-1, r_{i+1} is a successor of r_i in U_i'

[Def. 2.2-2] ⇒ there is a chain of nodes $r_1, r_2, \ldots, r_k$ such that

$r_1 = \Pi_1(p)$, $r_k = n$, and for i=1,...,k-1, r_{i+1} is a successor of r_1,

unless there is some j such that $r_j = m_1$ and φ_1 is an Update firing,

in which case $r_{j+1} = \Pi_1(q_1')$ where q_1' is the number-3 input to φ_1.

and there is still a chain $r_{j+1}, \ldots, r_k$ in which $r_k = n$ and for

i=j+1,...,k-1, r_{i+1} is a successor of r_i in U_1 [(15)+(9)+(11)+

(12)+(13)+(14)+Def. 2.2-2] ⇒ n is reachable in U_1 from either

$\Pi_1(p)$ or $\Pi_1(q_1')$ where q_1' is on an arc in $S_1 \cdot \Omega_1$ [Def. 2.2-2]

(19) For any pointer p, p is on an arc in $S_1 \cdot \Omega_1$ or $\Pi_1(p)$ is in an

ordered pair in $SM_1(n)$ for some node n∈N_1 ⇒ p∈dom Π_1

Thm. 2.2-1+Def. 2.2-1

Let n be any node equal to or reachable from $\Pi_1'(p_1)$ in U_1'. Then n is

equal to or reachable in U_1 from some node $\Pi_1(p_1')$ where p_1' is some

pointer on an arc in $S_1 \cdot \Omega_1$ [(16)+(19)+(18)]. n≠m_1 or φ_1 is not an

Assign, Update, or Delete ⇒ $SM_2'(I(n)) = I(SM_1'(n))$ [(10)]. n = m_1 and

φ_1 is an Assign, Update, or Delete ⇒ $SM_2'(I(n)) = I(SM_1'(n))$ [(11)+(12)+

(13)+(14)]. Therefore, $U_2' \cdot \Pi_2'(p_2) \stackrel{I}{=} U_1' \cdot \Pi_1'(p_1)$ [(17)]. From this and

(4)+(5)+(6)+(16),

(20) For any arc b which is not an output arc of d and has a token in

$S_1 \cdot \Omega_1 \varphi_1$, Match($(b, S_2 \cdot \Omega_2 \varphi_2)$, I, $(b, S_1 \cdot \Omega_1 \varphi_1)$)

(21) For each arc b which is an output arc of d and has a token in

$S_1 \cdot \Omega_1 \varphi_1$, the value $v_i = TV(b, S_1 \cdot \Omega_1 \varphi_1)$, for i=1,2, is output by φ_i

(4)

(22) d is a pI operator $\Rightarrow$ there is an input arc a of d such that

$\quad$ TV$(a, S_1 \cdot \Omega_1) = v_1$ $\hspace{4cm}$ (3)+(21)+Def. 2.2-4

(23) d is a pI operator $\wedge$ v_1 is non-pointer $\Rightarrow v_2 = v_1$ $\hspace{3cm}$ (6)

(24) d is a pI operator $\wedge$ v_1 is a pointer $\Rightarrow I(\Pi_1'(v_1)) = \Pi_2'(v_2)$ [(7)+(15)]

$\quad$ $\wedge$ since v_1 is on an arc in $S_1 \cdot \Omega_1$, for any node n equal to or

$\quad$ reachable from $\Pi_1'(v_1)$ in U_1', n equals or is reachable from $\Pi_1(v_1)$

$\quad$ in U_1 [(21)+(15)+(19)+(18)], so $SM_2'(I(n)) = I(SM_1'(n))$ [(10)] $\Rightarrow$

$\quad$ $U_2' \cdot \Pi_2'(v_2) \overset{I}{=} U_1' \cdot \Pi_1'(v_1)$

(25) v_1 is non-pointer and d is not a structure or pI operator $\Rightarrow v_1$

$\quad$ depends only on the type of actor d is and on the values on d's

$\quad$ input arcs in $S_1 \cdot \Omega_1$, for i=1,2 $\wedge$ all those input arcs hold non-

$\quad$ pointer values $\Rightarrow v_1 = v_2$ [(6)]

(26) v_1 is non-pointer and is not identically zero, and d is a

$\quad$ structure operator $\Rightarrow v_1$ depends only on the type of d, the non-

$\quad$ pointer input to φ_1, and the value and set of selectors in

$\quad$ $SM_1(\Pi_1(q_i))$, for i=1,2 [(8)] $\Rightarrow v_1 = v_2$ [(6)+(11)]

(27) v_1 is not a pointer $\Rightarrow v_1 = v_2$ $\hspace{3cm}$ (22)+(23)+(25)+(26)

(28) v_1 is a pointer and d is not a pI operator $\Rightarrow$ d is a Select or Copy

(29) φ_1 is a Select firing with selector input s $\Rightarrow v_1 = q_1'$, where the

$\quad$ pair $(s, \Pi_1(q_1'))$ is in $SM_1(m_1)$ [(8)] $\Rightarrow \varphi_2$ is a Select firing with

$\quad$ selector input s and $(s, I(\Pi_1(q_1')))$ is in $SM_2(m_2)$ [(6)+(11)] $\Rightarrow$

$\quad$ $v_2 = q_2'$ where $\Pi_2(q_2') = I(\Pi_1(q_1')) \wedge$ every node reachable from $\Pi_1'(q_1')$

$\quad$ in U_1' is reachable in U_1 from $\Pi_1(q_1')$, hence from m_1 [(15)+(19)+

$\quad$ (18)+(8)+Def. 2.2-2] $\Rightarrow$ since q_1 is on an arc in $S_1 \cdot \Omega_1$, for every

$\quad$ node n equal to or reachable from $\Pi_1'(v_1)$ in U_1',

$\quad$ $SM_2'(I(n)) = I(SM_1'(n))$ [(8)+(10)] $\Rightarrow U_2' \cdot \Pi_2'(v_2) \overset{I}{=} U_1' \cdot \Pi_1'(v_1)$

(30) φ_1 is a Copy firing $\Rightarrow \exists (q_1', n_1): \Pi_1' = \Pi_1 \cup \{(q_1', n_1)\}$, $v_1 = q_1'$, and

$SM_1'(n_1) = SM_1(m_1)$ $[(21)+(8)] \wedge \varphi_2$ is a Copy firing $\Rightarrow \exists (q_2', n_2):$

$\Pi_2' = \Pi_2 \cup \{(q_2', n_2)\}$, $v_2 = q_2'$, and $SM_2'(n_2) = SM_2(m_2)$ $[(1)+(8)] \Rightarrow$

letting $I' = I \cup \{(n_1, n_2)\}$, $I'(\Pi_1'(v_1)) = \Pi_2'(v_2) \wedge$ since every

successor of n_1 in U_1' is a successor of m_1 in U_1 and no node's

content is changed, each node reachable from n_1 in U_1' is reachable

from m_1 in U_1 $[(9)+$Def. 2.2-2$] \Rightarrow$ letting m be any node reachable

from n_1 in U_1', $m \in N_1$, so $SM_2'(I'(m)) = SM_2'(I(m)) = I(SM_1'(m)) =$

$I'(SM_1'(m))$ $[(8)+(10)] \wedge SM_2'(I'(n_1)) = SM_2'(n_2) = SM_2(m_2) =$

$I(SM_1(m_1)) = I'(SM_1'(n_1))$ $[(8)+(7)] \Rightarrow U_2' . \Pi_2'(v_2) \overset{I'}{=} U_1' . \Pi_1'(v_1)$

Therefore, v_1 is a pointer and d is not a pI operator $\Rightarrow$

$U_2' . \Pi_2'(v_2) \overset{I'}{=} U_1' . \Pi_1'(v_1)$, where

$$I' = \begin{cases} I & \text{if } \varphi_1 \text{ is not a Copy firing} \\ I \cup \{(n_1, n_2)\} & \text{if } \varphi_1 \text{ is a Copy firing } (d, (q_1', n_1)) \text{ and} \\ & \varphi_2 \text{ is a Copy firing } (d, (q_2', n_2)) \end{cases}$$

$[(30)+(8)+(7)]$. In summary, then, for each arc b in P,

Match$((b, S_2 \cdot \Omega_2 \varphi_2),$ I', $(b, S_1 \cdot \Omega_1 \varphi_1))[(4)+(20)+(21)+(27)+(22)+(24)]$; i.e.,

$S_2 \cdot \Omega_2 \varphi_2$ equals $S_1 \cdot \Omega_1 \varphi_1$ [Def. 2.4-3].

$\triangle$

<u>Lemma 5.3-4</u> Let S be any initial state for an L_{BS} program P, and let the

heap in S be (N, Π, SM). Let Ω be any firing sequence starting in S, let ω

be $\eta(S, \Omega)$, and let e be any execution of any structure operation (except

Copy). Let p be $V(Ent(e, 1))$, let q be the unique pointer in dom Π such

that $DD_\omega(q, p)$, and let $n = \Pi(q)$. Then the conclusions depicted in

Table 5.3-1 can be drawn about the values of e's output entries in ω.

Proof:

Key definitions: Def. 2.2-5 - structure operations; Def. 4.2-6 - initia-
tion; Def. 5.1-8 - reach; Def. 5.1-9 - dynamic descendancy;
Def. 5.2-7 - heap determined by a computation

(1) Let e be $Ex(d,k)$ and let $Int(P)$ be (St, I, IE). Then $d \in St\text{-}DL$

$$\text{Defs. } 4.3\text{-}2+4.3\text{-}1$$

Let f be any entry such that $T(f)$ has source $Src(e,i)$ for any i. Then
there is a prefix $\theta\varphi$ of Ω, containing exactly k firings of d, such that
tokens of value $V(f)$ appear in the number-i group of output arcs of d at
the transition from $S \cdot \theta$ to $S \cdot \theta\varphi$ [(1)+Lemma 4.3-1]. Therefore,

(2) φ must be the kth firing of d in Ω Def. 2.1-5

(3) Let α be $\eta(S,\theta)$ and let NAR be the node activation record derived
   from θ and α. Then the heap determined by α from the heap in S
   and NAR, $(N_\alpha, \Pi_\alpha, SM_\alpha)$, is defined and is identical to the heap in
   $S \cdot \theta$ Thm. 5.2-1

(4) Let β be $\eta(S,\theta\varphi)$. Then $g = Ent_\beta(e,1)$ is the first entry in β which
   is not in α, p is the number-1 input to φ, and there are m input
   entries to e in β, where φ removes m tokens   (1)+(2)+Alg. 4.3-1

(5) The value of $Src(e,i)$ in ω equals $V(f)$, and that depends on
   $SM_\alpha(\Pi_\alpha(p))$ as in Table 2.2-1   (2)-(4)+Def. 4.2-6

(6) $m = In(I(d))$, so e is initiated in β (4)+Defs. 4.3-2+4.3-1

(7) Let NAR' be the node activation record derived from $\theta\varphi$ and β, and
   let CC_α and CC_β be the Creating-Copy functions corresponding to
   NAR and NAR' respectively. Then p is the value in β of the output
   entries of a Copy execution C or $p \in dom \Pi = CC_\beta(p)$ is defined, the
   first entry in β with value p is an output entry of $CC_\beta(p)$, that

entry is strictly preceded by $Ent_\beta(CC_\beta(p),1)$, and no other Copy

execution has output entries of value p, so $C = CC_\beta(p)$ [(4)+

Lemma 5.2-3] $\Rightarrow$ there is a node m such that $NAR'(C) = (p,m)$

[Def. 5.2-5] $\wedge$ since $V(g) = p$, $Ent_\beta(C,1)$ precedes g — i.e., is in

α —— so C is initiated in α [(4)+Def. 4.3-1] $\Rightarrow NAR(C) = (p,m)$

[(3)+Lemma 5.2-5] $\Rightarrow CC_\alpha(p) = CC_\beta(p) = C$ [Def. 5.2-5]

Dynamic descendancy relations in a computation depend only on the input

and output entries of Copy executions in that computation. Furthermore,

(8) α and β are prefixes of ω, so every Copy execution which has input

or output entries in α or β has the same input or output entries

in ω [(3)+(4)+Alg. 4.3-1]. Hence

(9) $p \notin dom\ \Pi \Rightarrow$ letting q' be the unique pointer in dom Π such that

$p' = V(Ent_\alpha(CC_\alpha(p),1))$ is dynamically descended from q' in α,

$DD_\omega(q',p')$ and p' is the value of $Ent_\omega(CC_\alpha(p),1)$. Also, p is the

value in β, hence in ω, of the output entries of $CC_\alpha(p)$ [(7)+(8)], so

$DD_\omega(q',p)$, and since q is unique in dom Π, q' = q

Since $DD_\omega(p,p)$,

(10) If $p \in dom\ \Pi$, then q = p, otherwise q is the unique pointer in dom Π

such that $V(Ent_\alpha(CC_\alpha(p),1))$ is dynamically descended from q in α [(9)]

(11) α and β are prefixes of ω, and α, β, and ω are causal computations

for Int(P)   (8)+(4)+Lemma 4.3-2

(12) For any Update or Delete execution U, D(U) extends to the end of

$H_p^\alpha \Rightarrow$ U is initiated in $\alpha \Rightarrow Ent_\beta(U,2)$ is in α; i.e., U has the

same selector input in ω and α (11)+Def. 5.2-6

e is a Fetch or Assign execution and is in no reach in $\omega \Rightarrow$ e is in no

reach in β [(11)+(7)+(6)+Lemma 5.2-6] $\Rightarrow Ent_\omega(e,1)$ is not in the duration

of any Assign execution in β [(11)+Def. 5.1-6] $\Rightarrow$ no Assign execution duration extends to the end of H_p^α [(3)+(4)+Lemma 5.2-7] $\Rightarrow$ the value in $SM_\alpha(\Pi_\alpha(p))$ is the value in $SM(\Pi(q))$ [(3)+(10)] $\Rightarrow$ the value of $Src(e,i)$ is as given in Table 5.3-1 [(5)].

e is a Select, Update, or Delete execution and is in no reach in ω $\Rightarrow Ent_\beta(e,1)$ is not in the duration of any Update or Delete execution which has selector input s in ω [(11)+(7)+(6)+Lemma 5.2-6] $\Rightarrow$ there is no Update or Delete execution with selector input s in α whose duration extends to the end of H_p^α [(12)+(3)+(4)+Lemma 5.2-7] $\Rightarrow$ for any node m, the pair (s,m) is in $SM_\alpha(\Pi_\alpha(p))$ iff $(s,m) \in SM(\Pi(q))$ [(3)+(10)] $\Rightarrow$ the value of $Src(e,i)$ is as given in Table 5.3-1 [(5)].

(13) e is a First execution or a Next execution with selector input s $\Rightarrow$ the value of $Src(e,i)$ depends just on s and the set O of selectors in the ordered pairs in $SM_\alpha(\Pi_\alpha(p))$ [(5)]

(14) e is in the reach of an Update/Delete execution U in ω $\Rightarrow Ent_\beta(e,1)$ is in $D(U)$ in β [(11)+(7)+(6)+Lemma 5.2-6] $\Rightarrow D(U)$ extends to the end of H_p^α [(3)+(4)+Lemma 5.2-7] $\Rightarrow$ letting s be the selector input to U in α, hence in ω, if U is an Update, then $s \in S^c$ and $s \in O$, and if U is a Delete, then $s \in S^b$ and $s \notin O$ [(13)+(12)+(3)+(10)]

(15) For each $s \in \Sigma$, e is not in the reach of any Update/Delete execution with selector input s in ω $\Rightarrow$ the duration of no such execution extends to the end of H_p^α [(11)+(7)+(6)+(3)+(4)+(12)+Lemmas 5.2-6 +5.2-7] $\Rightarrow s \in O$ iff s is in a pair in $SM(\Pi(q))$ iff $s \in S^a$ [(3)+(10)]

Therefore, $s \in O$ iff $s \in (S^a - S^b) \cup S^c$ [(14)+(15)], so the value of $Src(e,i)$ depends on s and $(S^a - S^b) \cup S^c$ as in Table 5.3-1.

$\triangle$

__Lemma 5.3-5__ For any L_{BS} program P, let S_1 and S_2 be any two equal initial

standard states for P. For i=1,2, let Ω_i be any halted firing sequence

starting in S_i and let $\omega_i = \eta(S_i, \Omega_i)$. Then, given Int(P), the pair

consisting of ω_1 and ω_2 satisfies the Initial Structure Constraint and

the First/Next Output Constraint.

__Proof:__

(1) Let ρ be the equal pointer relation defined from Int(P). Then,

    since ω_1 and ω_2 are both computations for Int(P), ρ is defined

    for them [Lemma 4.3-2+Def. 5.1-10]

(2) There is a single one-to-one mapping I under which the conditions in

    S_1 and S_2 of each arc in P match   Def. 2.4-3

(3) For i=1,2, let the heap in S_i be (N_i, Π_i, SM_i). Let p_i and p_{i+2} be

    any two pointers such that neither is the value of an output entry

    of a Copy execution in ω_i. Then there is no $q_i \neq p_i$ such that

    $DD_{\omega_i}(q_i, p_i)$ and there is no $q_{i+2} \neq p_{i+2}$ such that $DD_{\omega_i}(q_{i+2}, p_{i+2})$

                                                                Def. 5.1-9

$(p_1, \omega_1)\rho(p_2, \omega_2) \Rightarrow$ for i=1,2, p_i is the value in ω_i of a source; i.e.,

p_i is the value of an entry in ω_i [Defs. 5.1-10+4.2-6] $\Rightarrow p_i \in dom\ \Pi_i$

[(3)+Lemma 5.2-3]. Similarly, $(p_3, \omega_1)\rho(p_2, \omega_2) \Rightarrow p_3 \in dom\ \Pi_1$ and $p_2 \in dom\ \Pi_2$,

and $(p_1, \omega_1)\rho(p_4, \omega_2) \Rightarrow p_1 \in dom\ \Pi_1$ and $p_4 \in dom\ \Pi_2$. Therefore, $(p_1, \omega_1)\rho(p_2, \omega_2)$

and $(p_3, \omega_1)\rho(p_2, \omega_2) \Rightarrow \Pi_2(p_2) = I(\Pi_1(p_1))$ and $\Pi_2(p_2) = I(\Pi_1(p_3))$ [(1)-(3)+

Thm. 5.3-2] $\Rightarrow \Pi_1(p_3) = \Pi_1(p_1)$ [(2)+(3)+Def. 2.2-1] $\Rightarrow p_3 = p_1$ [(3)+

Def. 2.2-1]. Also, $(p_1, \omega_1)\rho(p_2, \omega_2)$ and $(p_1, \omega_1)\rho(p_4, \omega_2) \Rightarrow$

$\Pi_2(p_2) = I(\Pi_1(p_1))$ and $\Pi_2(p_4) = I(\Pi_1(p_1))$ [(1)-(3)+Thm. 5.3-2] $\Rightarrow$

$\Pi_2(p_2) = \Pi_2(p_4) \Rightarrow p_2 = p_4$ [(2)+(3)+Def. 2.2-1]

(4) Let e_1 and e_2 be any two executions of structure operations initiated in ω_1 and ω_2 respectively. For $i=1,2$, let p_i be $V(Ent_{\omega_i}(e_i,1))$, let q_i be the unique pointer in dom Π_i such that $DD_{\omega_i}(q_i,p_i)$, and let $n_i = \Pi_i(q_i)$ Lemma 5.2-4

(5) $(p_1,\omega_1)\rho(p_2,\omega_2) \Rightarrow (q_1,\omega_1)\rho(q_2,\omega_2)$ [(1)-(4)+Thm. 5.3-2] $\Rightarrow$
$SM_2(n_2) = I(SM_1(n_1))$ [(1)-(4)+Thm. 5.3-2] $\Rightarrow$ the value and the sets
of selectors in $SM_2(n_2)$ and $SM_1(n_1)$ are identical [Def. 2.4-1]

(6) For $i=1,2$, e_i does not fall into a reach in ω_i $\Rightarrow$ if e_i is a Fetch
or Assign execution, then for $j=1,2$, the value of $Src(e_i,j)$ in ω_i
depends only on the value in $SM_i(n_i)$, if e_i is a Select, Update, or
Delete execution, then the value of the source $Src(e_i,2)$ in ω_i
depends only on $V(Ent_{\omega_i}(e_i,2))$ and the set of selectors in $SM_i(n_i)$,
and if e_i is an Update or Delete execution, the value of $Src(e_i,1)$
is identically zero [(1)-(4)+Lemma 5.3-4]

For $i=1,2$, e_i does not fall into a reach in ω_i and $(p_1,\omega_1)\rho(p_2,\omega_2)$ $\Rightarrow$
if e_1 and e_2 are two Fetch or Assign executions, then for $j=1,2$, the
values of $Src(e_1,j)$ in ω_1 and of $Src(e_2,j)$ in ω_2 are the same, if e_1 and
e_2 are each a Select, Update, or Delete execution, with $V(Ent_{\omega_1}(e_1,2))$
and $V(Ent_{\omega_2}(e_2,2))$ the same, then the values of $Src(e_1,2)$ in ω_1 and of
$Src(e_2,2)$ in ω_2 are the same, and if e_1 and e_2 are each an Update or
Delete execution, the values of $Src(e_1,1)$ in ω_1 and $Src(e_2,1)$ in ω_2 are
the same [(6)+(5)]. Therefore, the pair consisting of ω_1 and ω_2 satisfies
the Initial Structure Constraint [(1)+(4)+Const. 5.1-5].

(7) Assume e_1 and e_2 are two First executions or two Next executions with
$V(Ent_{\omega_1}(e_1,2)) = V(Ent_{\omega_2}(e_2,2)) = s$. Then, for $i=1,2$, for $j=1,2$,
the value of $Src(e_i,j)$ in ω_i depends only on s and on the set S_i

of selectors, defined by $S_1 = (S_1^a - S_1^b) \cup S_1^c$, where

$$S_1^a = \{s \in \Sigma \mid \exists m: (s,m) \in SM_1(n_1)\}$$

$$S_1^b = \{s \in \Sigma \mid \exists \text{Delete } D_1: e_1 \in R(D_1) \text{ in } \omega_1 \wedge s = V(\widetilde{Ent}_{\omega_1}(D_1, 2))\}$$

and $S_1^c = \{s \in \Sigma \mid \exists \text{Update } U_1: e_1 \in R(U_1) \text{ in } \omega_1 \wedge s = V(Ent_{\omega_1}(U_1, 2))\}$

$$(1)+(3)+(4)+\text{Lemma } 5.3-4$$

$(p_1, \omega_1) \rho (p_2, \omega_2) \Rightarrow S_1^a = S_2^a$ [(5)]. e_1 is in the reach of an Update (Delete) execution with selector input s in ω_1 iff e_2 is in the reach of an Update (Delete) execution with selector input s in $\omega_2 \Rightarrow S_1^b = S_2^b$ and $S_1^c = S_2^c$ [(6)]. Hence $(p_1, \omega_1) \rho (p_2, \omega_2)$ and e_1 is in the reach of an Update (Delete) execution with selector input s in ω_1 iff e_2 is in the reach of an Update (Delete) execution with selector input $s \Rightarrow S_1 = S_2$ [(6)] $\Rightarrow$ for $j=1,2$, the values of $Src(e_1, j)$ in ω_1 and $Src(e_2, j)$ in ω_2 are the same [(6)]. Therefore, the pair consisting of ω_1 and ω_2 satisfies the First/Next Output Constraint [(1)+(4)+Const. 5.1-6].

$\triangle$

__Lemma 5.3-11__ Let S_1 and S_2 be any two equal initial standard states for the same L_{BS} program P and let Ω_1 and Ω_2 be any two halted firing sequences starting in S_1 and S_2 respectively. Let $\omega_1 = \eta(S_1, \Omega_1)$ and $\omega_2 = \eta(S_2, \Omega_2)$, and assume that these are computations for Int(P). Let α_1 and α_2 be any two causal computations for Int(P) and let ρ be the equal pointer relation defined from Int(P). If given Int(P),

(1) for $i=1,2$, for any structure operation execution e, e is initiated in $\alpha_i \nRightarrow e$ is initiated in ω_i, for every integer j, if there is an entry $Ent_{\alpha_i}(e,j)$ in α_i, then there is an $Ent_{\omega_i}(e,j)$ in ω_i with the same value, and if there is an entry in α_i whose transfer has source

Src(e,j), then there is an entry in ω_1 with the same value whose
transfer has source Src(e,j),

(2) for i=1,2, for every structure operation execution e initiated in α_i
and any Assign, Update, or Delete execution A, $e \in R(A)$ in α_i iff
$e \in R(A)$ in ω_i, and

(3) for any pointers p_1 and p_2, $(p_1,\alpha_1)\rho(p_2,\alpha_2) \Rightarrow (p_1,\omega_1)\rho(p_2,\omega_2)$,
then α_i satisfies the Atomic Output, Structure Output, and Unique Pointer
Generation Constraints, and the pair consisting of α_1 and α_2 satisfies the
Initial Structure and the First/Next Output Constraints.

Proof:

(4) For i=1,2, ω_i satisfies the Atomic Output and Structure Output

    Constraints given Int(P)   Lemma 5.3-3

For each Fetch, Assign, Select, Update, or Delete execution e initiated
in α_i and any Assign, Update, or Delete execution A, $e \in R(A)$ in α_i $\Rightarrow$
$e \in R(A)$ in ω_i [(2)] $\wedge$ e is a Fetch, Assign, Select, Update, or Delete
execution initiated in ω_i, and for j=1,2, if there is an entry $Ent_{\alpha_i}(A,j)$
then there is an entry $Ent_{\omega_i}(A,j)$ and $V(Ent_{\alpha_i}(A,j)) = V(Ent_{\omega_i}(A,j))$ [(1)]
$\wedge$ for k=1,2, the value of Src(e,k) in ω_i (if any) depends on the actions
of A and e and possibly on $V(Ent_{\omega_i}(A,2))$ and $V(Ent_{\omega_i}(A,3))$, as in the
Constraints [(4)+Def. 4.2-6+Consts. 5.1-3+5.1-4] $\Rightarrow$ for k=1,2, the value
of Src(e,k) in α_i (if any) depends on the actions of A and e and possibly
on $V(Ent_{\alpha_i}(A,2))$ and $V(Ent_{\alpha_i}(A,3))$ as in the Constraints [(1)+Def. 4.2-6].
Therefore, α_i satisfies the Atomic Output and Structure Output Constraints
given Int(P).

(5) ω_i satisfies the Unique Pointer Generation Constraint given

Int(P) = (St, I, IE)   Lemma 5.3-6

Let C be any Copy execution initiated in α_i, and let p be the value of

its output entries in α_i (if any). Then α_i does not satisfy the Unique

Pointer Generation Constraint $\Rightarrow$ there is an execution e$\neq$C whose output

entries have value p in α_i and e either is in IE, is a Copy execution, or

is a Select execution not in a reach in α_i [Const. 5.1-7] $\Rightarrow$ C and e have

output entries of value p in ω_i [(1)] $\wedge$ by causality, e is initiated in

α_i [Def. 4.2-7] $\Rightarrow$ either e$\in$IE, e is a Copy execution not equal to C, or

e is a Select execution not in a reach in ω_i [(2)] $\Rightarrow$ ω_i does not satisfy

the Unique Pointer Generation Constraint [Const. 5.1-7]. Therefore, α_i

does satisfy that Constraint given Int(P) [(5)].

(6) The pair ω_1, ω_2 satisfies the Initial Structure and First/Next

    Output Constraints given Int(P)   Lemma 5.3-5

(7) For i=1,2, let p_i and p_{i+2} be two pointers such that neither is the

    value in α_i of an output entry of a Copy execution. Then for any

    q, $DD_{\alpha_i}(q,p_i) \Rightarrow q = p_i$ and $DD_{\alpha_i}(q,p_{i+2}) \Rightarrow q = p_{i+2}$ Def. 5.1-9

(8) $(p_1,\alpha_1)\rho(p_2,\alpha_2) \wedge (p_3,\alpha_1)\rho(p_2,\alpha_2) \Rightarrow p_1$, p_2, and p_3 are each the

    value of the output entries in α_i of an execution e which either

    is in IE or is a Select execution not in a reach in α_i

                                        (7)+Defs. 5.1-10+4.2-6

(9) $\wedge (p_1,\omega_1)\rho(p_2,\omega_2) \wedge (p_3,\omega_1)\rho(p_2,\omega_2)$ (3)

$\Rightarrow p_1$, p_2, and p_3 are each the value of the output entries in ω_i of an

execution e which either is in IE or is a Select execution not in a reach

(since e is initiated in α_i, by causality [Def. 4.2-7]) [(1)+(2)] $\Rightarrow$ none

of p_1, p_2, or p_3 is the value of the output entries of a Copy execution

in ω_1 or ω_2 [(5)+Const. 5.1-7] $\Rightarrow$

(10) $p_1 = p_3$ (9)+(6)+Const. 5.1-5

By symmetry,

(11) $(p_1, a_1) \rho (p_2, a_2) \land (p_1, a_1) \rho (p_4, a_2) \to p_2 = p_4$

(12) Let e_1 and e_2 be any two Fetch or two Assign executions initiated in a_1 and a_2 respectively with pointer inputs p_1 and p_2 such that $(p_1, a_1) \rho (p_2, a_2)$. Then e_1 and e_2 are two Fetch or two Assign executions initiated in ω_1 and ω_2 respectively with pointer inputs p_1 and p_2 such that $(p_1, \omega_1) \rho (p_2, \omega_2)$ [(1)+(3)]. For $i=1,2$, e_i does not fall into a reach in $a_i \to e_i$ does not fall into a reach in ω_i [(2)] $\to$ for $j=1,2$, the values of $Src(e_1, j)$ in ω_1 and $Src(e_2, j)$ in ω_2 are the same [(6)+Const. 5.1-5] $\to$ the values of $Src(e_1, j)$ in a_1 and $Src(e_2, j)$ in a_2 are the same [(1)+Def. 4.2-6]

(13) Let e_1 and e_2 be any two Select, Update, or Delete executions initiated in a_1 and a_2 with equal selector inputs and pointer inputs p_1 and p_2 such that $(p_1, a_1) \rho (p_2, a_2)$. Then e_1 and e_2 are two Select, Update, or Delete executions initiated in ω_1 and ω_2 with equal selector inputs [(1)] and pointer inputs p_1 and p_2 such that $(p_1, \omega_1) \rho (p_2, \omega_2)$ [(3)]

(14) e_1 does not fall into a reach in $a_1 \to e_1$ does not fall into a reach in ω_1 [(13)+(2)] $\to Src(e_1, 2)$ has the same value in ω_1 as $Src(e_2, 2)$ has in ω_2, and if both e_1 and e_2 are Update or Delete executions, then the values of $Src(e_1, 1)$ in ω_1 and $Src(e_2, 1)$ in ω_2 are the same [(13)+(6)+Const. 5.1-5] $\to Src(e_1, 2)$ has the same value in a_1 as $Src(e_2, 2)$ has in a_2, and if both e_1 and e_2 are Update or Delete executions, then the values of $Src(e_1, 1)$ in a_1 and $Src(e_2, 1)$ in a_2 are the same [(1)+Def. 4.2-6]

The pair α_1, α_2 satisfies the Initial Structure Constraint [(7)+(8)+(10)+ (11)+(12)+(13)+Const. 5.1-5].

(15) Let e_1 and e_2 be two First executions, or two Next executions with the same selector inputs, initiated in α_1 and α_2. Then e_1 and e_2 are two First executions, or two Next executions with equal selector inputs, initiated in ω_1 and ω_2 (1)

(16) Their pointer inputs are p_1 and p_2 such that $(p_1,\alpha_1)\rho(p_2,\alpha_2)$ and for each selector s, e_1 is in the reach of an Update (Delete) execution A_1 with selector input s in α_1 iff e_2 is in the reach of Update (Delete) execution A_2 with selector input s in α_2 $\Rightarrow$ their pointer input values are p_1 and p_2 [(1)] such that $(p_1,\omega_1)\rho(p_2,\omega_2)$ [(3)] and e_1 is in the reach of an Update (Delete) execution A_1 with selector input s in ω_1 iff e_2 is in the reach of an Update (Delete) execution A_2 in ω_2 [(15)+(2)+(1)] $\Rightarrow$ for j=1,2, the value of $Src(e_1,j)$ in ω_1 is the same as the value of $Src(e_2,j)$ in ω_2 [(15)+(6)+Const. 5.1-6] $\Rightarrow$ for j=1,2, the values of $Src(e_1,j)$ in α_1 and $Src(e_2,j)$ in α_2 are the same [(1)+Def. 4.2-6]

Therefore, the pair α_1, α_2 satisfies the First/Next Output Constraint [15)+(16)+Const. 5.1-6].

$\triangle$

Appendix E

Proofs from Chapter 7

<u>Theorem 7.1-1</u> Let S be any initial modified state from any L_{BS} program

P, and let S' be the corresponding initial standard state. Let Ω be any

firing sequence starting in S on the modified interpreter. Then

  A: Ω is also a firing sequence starting in S' on the standard

     interpreter, and

  B: $S' \cdot \Omega \mu S \cdot \Omega$.

<u>Proof</u>:

Key definitions: Def. 2.1-5 - standard interpreter; Def. 3.3-7 - Standard

functions; Def. 3.3-8 - Strip; Def. 3.3-9 - modified interpreter;

Def. 7.1-1 - congruency (μ)

Proof is by induction on the length of Ω.

Basis: $|\Omega| = 0$. This empty sequence is a firing sequence on any data-

flow interpreter starting in any initial state [Def. 2.3-1]. Furthermore,

$S' \cdot \Omega = S'$ and $S \cdot \Omega = S$ [Def. 2.3-1], so since Q is empty in an initial

state, $S' \cdot \Omega \mu S \cdot \Omega$ [Defs. 3.3-5+7.1-1]. Hence A and B.

Induction step: Assume A and B are true for an firing sequence of length

n, n $\geq$ 0, and consider $\Omega\varphi$, starting in S, of length n+1.

  (1) Let d be the actor in P of which the last firing φ is a firing. Then

      d is enabled in $S \cdot \Omega$ Def. 2.3-1

  (2) The input and output arcs of d in $S \cdot \Omega$ are configured as required

      for enabling per Def. 2.1-4, and if d is a Select, $\exists$p: d$\in$Q(p) in

      $S \cdot \Omega$ Def. 3.3-6

(3) $S' \cdot \Omega \mu S \cdot \Omega$ ind. hyp. B

(4) If d is a gate, then its control input in $S' \cdot \Omega$ (being non-pointer)

    is the same as in $S \cdot \Omega$ (3)

For any input arc b of d, b is empty in $S' \cdot \Omega \Rightarrow$ b is empty in $S \cdot \Omega$ [(3)]

$\Rightarrow$ d is a merge gate with a <u>true</u> (<u>false</u>) control input in $S \cdot \Omega$, and b is

the F (T) input arc of d [(2)+Def. 2.1-4] $\Rightarrow$ d is a merge gate with a

<u>true</u> (<u>false</u>) control input in $S' \cdot \Omega$, and b is the F (T) input arc of d [(4)].

For any output arc b of d, b is empty in $S \cdot \Omega$ [(2)+Def. 2.1-4] $\Rightarrow$ if b is

not empty in $S' \cdot \Omega$, then d is a Select and $\exists p$: $d \in Q(p)$ in $S \cdot \Omega$ [(3)] $\Rightarrow$

b is empty in $S' \cdot \Omega$ [(2)]. Therefore, d is enabled in $S' \cdot \Omega$ on the

standard interpreter [Def. 2.1-4], so

A: $\Omega \varphi$ is a firing sequence starting in S' on the standard interpreter

  [(1)+Def. 2.3-1]

(5) Let $S \cdot \Omega$ be (Γ_1, U_1, Q_1) and let Fire$(S \cdot \Omega, d)$ be (Γ_2, U_2, Q_2), while

    $S' \cdot \Omega$ is (Γ_1', U_1') and $S'' \cdot \Omega \varphi$ is (Γ_2', U_2'). Let Γ_s be

    Standard$_\Gamma((\text{Strip}(\Gamma_1, d), U_1), d)$

(6) $\Gamma_2' = $ Standard$_\Gamma((\Gamma_1', U_1'), d)$ and $U_2' = $ Standard$_U((\Gamma_1', U_1'), d)$ (5)

(7) $U_2 = $ Standard$_U((\text{Strip}(\Gamma_1, d), U_1), d)$ (5)

(8) For every arc b in P, the conditions of b in (Γ_1', U_1') and in

    (Γ_1, U_1, Q_1) match to within withheld outputs, and U_1' is identical

    to U_1 (3)+Def. 7.1-1

(9) Let b be any arc in P which is neither an input nor output arc of

    d. Then b's condition in Γ_2 is identical to b's condition in Γ_s

    which is identical to b's condition in Strip(Γ_1, d) [(5)] which is

    identical to b's condition in Γ_1, and b's condition in Γ_2' is

    identical to b's condition in Γ_1' [(6)].

(10) If b is a data output arc of a Select operator S, then $S \neq d$ [(9)+ Def. 2.1-1] $\wedge \forall p$: $S \in Q_2(p) \rightarrow [S \in Q_1(p) \vee S = d]$, so for all p, $S \in Q_2(p)$ iff $S \in Q_1(p)$

(11) For any arc b in P which is neither an input nor an output arc of d, the conditions of b in Fire($S' \cdot \Omega, d$) and in $S' \cdot \Omega \phi$ match to within withheld outputs [(8)+(9)+(10)+(5)+Def. 7.1-1]

(12) Let b be any arc which is an input arc of d and is not an output arc of d. Then b is not the T (F) input arc of a merge gate d with a <u>false</u> (<u>true</u>) control input in Strip(Γ_1,d) $\rightarrow$ b is not the T (F) input arc of a merge gate d with a <u>false</u> (<u>true</u>) control input in Γ_1 $\rightarrow$ b is not the T (F) input arc of a merge gate d with a <u>false</u> (<u>true</u>) control input in Γ_1' [(4)+(5)] $\rightarrow$ b is empty in Γ_s [(5)] and b is empty in Γ_2' [(5)+(6)] $\rightarrow$ b is empty in Γ_2 and in Γ_2'

(13) b is the T (F) input arc of a merge gate which has a <u>false</u> (<u>true</u>) control input in Γ_1 $\rightarrow$ b's condition in Γ_2' matches its condition in Γ_1' [(4)+(6)] $\wedge$ since d is a pI actor, b's condition in Strip(Γ_1,d) matches that in Γ_1 and d is a merge gate which has a <u>false</u> (<u>true</u>) control input in Strip(Γ_1,d) [Def. 2.2-4] $\rightarrow$ b's condition in Γ_s matches b's condition in Γ_1 [(5)] $\rightarrow$ since Γ_2 differs from Γ_s only in the conditions of d's output arcs, b's condition in Γ_2 matches b's condition in Γ_1 [(12)]

(14) b is a data output arc of a Select operator S $\rightarrow$ $S \neq d$ [(12)] $\wedge$ for all p, $S \in Q_2(p)$ iff [$S \in Q_1(p) \vee S = d$], so for all p, $S \in Q_1(p)$ iff $S \in Q_2(p)$

(15) b is a data output arc of a Select operator S $\rightarrow$ $\exists \rho$: $S \in Q_2(p) \rightarrow$ $S \in Q_1(p)$ [(14)] $\rightarrow$ b is empty in Γ_1 and has a token of value p in Γ_1'

[(8)] ⇒ b is the T (F) input arc of a merge gate d with a <u>false</u>

(<u>true</u>) control input in Γ_1 [(11)+(2)+Def. 2.1-4] ⇒ b is empty in

Γ_2 and has a token of value p in Γ_2' [(13)]

(16) b is not a data output arc of a Select operator S such that ∃p:

$S \in Q_2(p)$ ⇒ b is not a data output arc of a Select operator S such

that ∃p: $S \in Q_1(p)$ [(14)] ⇒ either b is empty in Γ_1 and Γ_1', or b

has tokens of non-pointer value v in Γ_1 and Γ_1', or b has a token

of pointer value p in Γ_1' and a token of value (p,R) or (p,W) in Γ_1

[(8)] ⇒ either [b is the T (F) input arc of a merge gate d with a

<u>false</u> (<u>true</u>) contol input in Γ_1 and either b is empty in Γ_2 and

Γ_2', b has a token of non-pointer value v in Γ_2 and Γ_2', or b has a

token of value p in Γ_2' and one of value (p,R) or (p,W) in Γ_2] [(13)]

or [b is not the T (F) input arc of a merge gate d with a <u>false</u>

(<u>true</u>) control input in Γ_1 and b is empty in Γ_2 and Γ_2'] [(11)]

(17) For any input arc b of d which is not an output arc of d, the

conditions of b in Fire($S \cdot Q$,d) and $S' \cdot Q_\varphi$ match to within withheld

outputs   (15)+(16)+(8)

For any output arc b of d, there are two cases to consider: d either is

or is not a structure operator.

Case I: d is not a structure operator

(18) Since d is not a Select, b is empty in Γ_2 iff b is empty in Γ_1 [(5)]

iff d is a T- (F-) gate which has a <u>false</u> (<u>true</u>) control input in

Strip(Γ_1,d) [(5)] iff d is a T- (F-) gate which has a <u>false</u> (<u>true</u>)

control input in Γ_1 iff d is a T- (F-) gate which has a <u>false</u> (<u>true</u>)

control input in Γ_1' [(4)] iff b is empty in Γ_2' [(6)]

If b has a token in Γ_2 and Γ_2', there are two sub-cases to consider.

Case Ia: d is a pI operator.

Let v be the value of the token on b in Γ_2'. Then there is an input arc a of d which holds a token of value v in Γ_1' that is removed by φ [Def. 2.2-4]. If d is a gate, it has the same control input in Γ_1 as in Γ_1', and since d is enabled, there is a token on a in Γ_1 [(4)+(2)+Def. 2.1 Def. 2.1-4]. The value of that token is v, if v is not a pointer, or (v,R) or (v,W), if v is a pointer [(8)]. There is a token of some value v' on a in Strip(Γ_1,d), and if d is a gate, it has the same control input in Strip(Γ_1,d) as in Γ_1'. Thus there is a token of value v' on b in Γ_s [(5)], so there is a token of value v, if v is not a pointer, or (v,R) or (v,W), if v is a pointer, on b in Γ_2.

Case Ib: d is not a pI actor (or a structure operator)

The value of the token on b in Γ_2 equals the value of the token on b in Γ_s, which depends only on the values on d's input arcs in Strip(Γ_1,d) and the type of actor d is, and the value of the token on b in Γ_2' depends, in exactly the same way, on the values on d's input arcs in Γ_1' and the type of actor d is [(5)+(6)]. The values on d's input arcs in both Γ_1 and Γ_1' are all non-pointers, as are the values on b in Γ_2 and Γ_2' [Def. 2.2-5]. The values on d's input arcs in Strip(Γ_1,d) are identical to those in Γ_1, which are identical to those in Γ_1' [(8)+(2)]. Therefore, the values on b in Γ_2 and Γ_2' are identical non-pointers.

In either case,

(19) If d is not a structure operator, then the conditions of any output
    arc b of d in Fire($S \cdot \Omega$,d) and $S' \cdot \Omega\varphi$ match to within withheld outputs
    [(18)+(5)] and $U_2 = U_1 = U_1' = U_2'$ [(7)+(8)+(6)]

Case II: d is a structure operator

(20) U_2 and the token on b in Γ_s depend only on U_1, the values on d's

input arcs in $Strip(\Gamma_1,d)$, and the pointer-node pair (p,n) in φ,

if d is a Copy. U_2' and the token on b in Γ_2' depend in exactly the

same way on U_1', the values on d's input arcs in Γ_1', and on (p,n),

if d is a Copy   (6)+(7)+Def. 2.3-1

There are tokens on all of d's input arcs in Γ_1 [(2)+Def. 2.1-4], so

the values on d's input arcs in Γ_1 and Γ_1' differ by at most an "R" or

"W" tag [(8)]. Thus, the values on d's input arcs in $Strip(\Gamma_1,d)$ and

Γ_1' are identical. Therefore, U_2 is identical to U_2' and b has identical

tokens in Γ_s and Γ_2'. b is not any output arc of a Copy or a data output

arc of a Select $\Rightarrow$ the tokens on b in Γ_s and Γ_2 have identical non-pointer

values $\Rightarrow$ the tokens on b in Γ_2 and Γ_2' have identical non-pointer values.

b is any Copy output arc $\Rightarrow$ the tokens on b in Γ_s and Γ_2' both have as value

a pointer p [Def. 2.3-1] $\Rightarrow$ the token on b in Γ_2 has value (p,R) or (p,W)

and the token on b in Γ_2' has value p. b is a data-output arc of a Select

d $\Rightarrow$ the tokens on b in Γ_s and Γ_2' both have a pointer value q [(5)+(6)]

$\Rightarrow$ b is empty in Γ_2 and $d \in Q_2(p)$. Therefore,

(21) If d is a structure operator, then U_2 is identical to U_2', and the

conditions of b in $Fire(S \cdot \Omega,d)$ and $S' \cdot \Omega \varphi$ match to within withheld

outputs   (8)

In either Case I or Case II, then,

(22) U_2 is identical to U_2' and the conditions of b in $Fire(S \cdot \Omega,d)$ and

$S' \cdot \Omega \varphi$ match to within withheld outputs   (19)+(21)

(23) For any arc b in P, the conditions of b in $Fire(S \cdot \Omega,d)$ and $S' \cdot \Omega \varphi$

match to within withheld outputs, and U_2 is identical to U_2' [(11)+

(17)+(22)]

(24) Let $S \cdot \Omega\varphi$ = Release$((\Gamma_2, U_2, Q_2))$ be (Γ_3, U_3, Q_3) [(5)]. Then U_3 is

identical to U_2, which is identical to U_2' (23)

(25) For any arc b in P, b is the data output arc of a Select S and $\exists p$:

$S \in Q_3(p) \rightarrow$ b is an output arc of a Select S and $\exists p$: $S \in Q_2(p) \rightarrow$ b is

empty in Γ_2 and there is a token of value p on b in Γ_2' [(23)+(5)]

$\rightarrow$ b is empty in Γ_3 and there is a token of value p on b in Γ_2'.

(26) b is not the data output arc of a Select S such that $\exists p$: $S \in Q_3(p) \rightarrow$

either b is not the data output arc of a Select S such that $\exists p$:

$S \in Q_2(p)$, implying that the condition of b in Γ_3 is identical to

that in Γ_2, or b is the data output arc of a Select S such that

$\exists p$: $S \in Q_2(p)$, which implies that b has a token of value (p,R) in

$\Gamma_3 \rightarrow$ either [b is empty in Γ_3 and Γ_2' or b has a token of non-

pointer value in Γ_3 and Γ_2' or b has a token of pointer value p in

Γ_2' and a token of value (p,R) or (p,W) in Γ_3] [(23)] or [$\exists p$: b has

a token of value p in Γ_2' and a token of value (p,R) in Γ_3] [(23)].

Therefore, for every arc b, the conditions of b in $S \cdot \Omega\varphi$ and $S' \cdot \Omega\varphi$ match

to within withheld outputs [(25)+(26)], so from this and [(24)],

$S \cdot \Omega\varphi \mu S' \cdot \Omega\varphi$.

$\triangle$

Theorem 7.1-3 Let P be any L_{BS} program. For any initial modified state

S for P, let S' be the corresponding initial standard state and let Ω be

any halted firing sequence starting in S. Then there is a halted firing

sequence Ω' which has Ω as a prefix such that $\eta(S', \Omega')$ is SOE-inclusive

of $\eta(S, \Omega)$.

Proof: Ω is a firing sequence starting in S' and $S' \cdot \Omega \mu S \cdot \Omega$ [Thm. 7.1-1],
so

(1) Ω is a prefix of a halted firing sequence Ω' starting in S'

Def. 2.3-1

(2) Let $\omega = \eta(S, \Omega)$ and $\omega' = \eta(S', \Omega')$. Let (Int, J) be the expansion for
P from $EE(L_{BS}, M)$ and let (Int', J') be the expansion for P from
$EE(L_{BS}, S)$. Then $Int' = Int = Int(P)$ Def. 4.3-2

(3) Let $Int = (St, I, IE)$. Then ω and ω' are both causal computations
for Int (2)+Lemma 4.3-2

(4) $\omega \in J_{S, \Omega}$, $\omega' \in J_{S', \Omega'}$, $\Phi(\omega)$ is the reduction of Ω, and $\Phi(\omega')$ is the
reduction of Ω' (2)+Lemma 4.3-3

(5) Let $e = Ex(d, k)$ be any execution in which $I(d)$ is a structure
operation. Then $d \in St-DL$ Def. 4.3-2

(6) e is initiated in $\omega \Rightarrow$ there are at least k firings of d in Ω
$[(2)-(5)+Thm. 4.3-2] \Rightarrow$ there are at least k firings of d in Ω' [(1)]
$\Rightarrow e$ is initiated in ω' [(1)-(5)+Thm. 4.3-2]

(7) Let NDE be the set of executions $NDE = \{Ex(d, k) \mid d \in St-DL\}$. Then
for any $e \in NDE$ which is initiated in ω, the initiating entry to e
is preceded in both ω and ω' by the initiating entries to exactly
$k-1$ other executions of d (3)+(1)+(4)+(6)+Cor. 4.3-1

Since the reduction of a prefix of Ω' is a prefix of the reduction of Ω'
[Def. 2.4-5], $\Phi(\omega)$ is a prefix of $\Phi(\omega')$ [(4)+(1)], so

(8) For any $n \leq |\Phi(\omega)|$, the nth execution from NDE to initiate in ω is
$Ex(d, k)$ iff the nth firing in $\Phi(\omega)$ is the kth firing of d [(7)+
Def. 4.3-4] iff the nth firing in $\Phi(\omega')$ is the kth firing of d iff
the nth execution from NDE to initiate in ω' is $Ex(d, k)$ [(7)+

Def. 4.3-4]

(9) Let e and e' be any two distinct executions of structure operations
such that e is initiated in ω. Then both e and e' are in NDE

(7)+Def. 4.3-2

There is an $n \le |\Phi(\omega)|$ such that e is the nth execution from NDE to
initiate in both ω and ω' [(9)+(8)], so

(10) e' initiates before e in ω iff e is the nth execution from NDE to
initiate in ω, for $n \le |\Phi(\omega)|$, e' is the mth, and $m < n$ iff e is
the nth execution from NDE to initiate in ω', $n \le |\Phi(\omega)|$, e' is
the mth, and $m < n$ [(8)] iff e' initiates before e in ω'

(11) Let $C = Ex(d,k)$ be any Copy execution initiated in ω. Then there
is one input entry to e in ω [Defs. 4.2-6+4.3-1+2.2-5], and there
are at least k firings of d in Ω [(9)+(7)+Lemma 4.3-1], so C has
output entries in $\omega = \eta(S,\Omega)$ [(2)+Lemma 7.1-2]

(12) Let $\theta\varphi$ be any prefix of Ω. Then θ and $\theta\varphi$ are both firing sequences
starting in S Def. 2.3-1

By Thm. 7.1-1, then, θ and $\theta\varphi$ are both firing sequences starting in S',
and $S' \cdot \theta \mu S \cdot \theta$, so

(13) for each arc b in P which holds a token in $S \cdot \theta$, b holds a token in
$S' \cdot \theta$, and the value of the token on b in $S' \cdot \theta$ is v iff the value of
the token on b in $S \cdot \theta$ is v, if v is not a pointer, or (v,R) or
(v,W), if v is a pointer [Def. 7.1-1], and

(14) if φ is a firing of a gate actor d, then d is enabled in both $S \cdot \theta$
and $S' \cdot \theta$ [(12)+Def. 2.3-1], so it has control tokens in both
states [Def. 2.1-4] whose values must be the same

(15) For any prefix Δ of Ω, Δ is a firing sequence starting in S and

in S', and $S' \cdot \Delta \mu S \cdot \Delta$ [Def. 2.3-1+Thm. 7.1-1]. Thus, for any arc b

of P, b holds a token in $S \cdot \Delta \Rightarrow$ b holds a token in $S' \cdot \Delta$ [Def. 7.1-1]

so $\text{Source}(b, S', \Delta) = \text{Source}(b, S, \Delta)$ [Lemma 7.1-3]

(16) Let $e = \text{Ex}(d,k)$ be any execution such that $I(d) \neq OA$ and there are

input entries to e in ω. Then $\text{In}(I(d)) > 0$ (3)+Def. 4.2-6

$I(d) \neq IG$, so $e \notin IE$ and $d \in \text{St-DL}$ [(3)+Defs. 4.3-1+4.3-2]. Hence,

(17) there is a prefix $\theta\varphi$ of Ω in which φ is the kth firing of d

                               (2)+Lemma 4.3-1

(18) $\theta\varphi$ is a prefix of Ω' in which φ is the kth firing of d   (17)+(1)

If d is a gate, it has the same control input in both $S \cdot \theta$ and $S' \cdot \theta$ [(17)+

(12)+(14)], so for each input arc b of d, there is a token on b in $S \cdot \theta$

but not in $S \cdot \theta\varphi$ iff there is a token on b in $S' \cdot \theta$ but not in $S' \cdot \theta\varphi$

[Def. 2.1-5]. Therefore,

(19) For any integer j, source s, and value v, there is an entry f in ω

such that $V(f) = v$ and $T(f)$ has source s and destination $\text{Dst}(e,j)$

iff, in going from $S \cdot \theta$ to $S \cdot \theta\varphi$, a token of value v, if v is not a

pointer, or (v,R) or (v,W), if v is a pointer, is removed from b,

the number-j input arc of d, and $s = \text{Source}(b, S, \theta)$ [(17)+Alg. 4.3-1]

iff in going from $S' \cdot \theta$ to $S' \cdot \theta\varphi$, a token of value v is removed from

b [(13)] and $s = \text{Source}(b, S', \theta)$ [(15)] iff there is an entry g in

ω' such that $V(g) = v$ and $T(g)$ has source s and destination

$\text{Dst}(e,j)$ [(18)+(16)+Alg. 4.3-1]

(20) Let $e = \text{Ex}(d,k)$ be any non-pI execution. Then $I(d) \neq OA$ [Def. 5.1-2],

so for any j, if there is an entry $\text{Ent}_\omega(e,j)$ in ω, then there is an

entry $\text{Ent}_{\omega'}(e,j)$ in ω' with the same value [(16)+(19)]

(21) Let f be <u>any</u> entry in ω. Let V(f) be v and let T(f) be

$(s,Dst(Ex(d,k),j))$ where $s = Src(e,i)$. Then $Ex(d,k) \in IE \to I(d) = IG$

$\to In(I(d)) = 0$ [Defs. 4.3-2+4.3-1]. Since $Ex(d,k)$ has an input

entry in ω, $In(I(d)) > 0$ [(3)+Def. 4.2-6] $\to Ex(d,k) \notin IE$. Therefore,

(22) if $I(d) \neq OA$, then there is an entry g in ω' such that $V(g) = v$ and

T(g) has source s [(16)+(19)]

(23) $I(d) = OA \to$ there is a token on an arc b in $S \cdot \Omega$ whose value is v, if

v is not a pointer, or (v,R) or (v,W) if v is a pointer, and

$s = Source(b,S,\Omega)$ [(21)+Alg. 4.3-1] $\to$ there is a token of value v

on b in $S' \cdot \Omega$ [(13)] $\to$ that token either is or is not removed by a

firing in Ω' which is not in Ω [(1)]

(24) Given any arc b, let θ be any prefix of Ω' longer than Ω such that

every firing in θ which removes a token from b is in Ω. Then for

any prefix Δ of Ω' such that $|\Omega| \leq |\Delta| \leq |\theta|$, there is a token on

b in $S' \cdot \Delta$ (23)

b is in the number-i group of output arcs of actor d' $\to$ for no Δ such

that $|\Omega| \leq |\Delta| \leq |\theta|$ is d' enabled in $S' \cdot \Delta$ [(24)+Def. 2.1-4] $\to$ there are

the same number of firings of d' in θ as in Ω [Def. 2.3-1]. Hence,

(25) $Source(b,S',\theta) = Source(b,S,\Omega) = s$ (24)+Def. 2.3-1+Lemma 7.1-3

(26) There is a token of value v on b in $S' \cdot \Omega$ which is removed by a

subsequent firing in $\Omega' \to$ there is a prefix $\theta\varphi$ of Ω' longer than Ω

such that every firing in θ which removes a token from b is in Ω

and φ removes a token from b $\to$ there is an entry g in ω' such that

$V(g) = v$ and T(g) has source $Source(b,S',\theta) = s$ (24)+(25)+Alg. 4.3-1

(27) There is a token of value v on b in $S' \cdot \Omega$ which is not removed by a

subsequent firing $\to$ there is a token of value v on b in $S' \cdot \Omega'$, Ω'

is halted, and Ω' is a prefix of Ω' longer than Ω in which every

firing which removes a token from b is in Ω [(1)] $\Rightarrow$ there is an

entry g in ω' such that $V(g) = v$ and $T(g)$ has source Source(b,S',Ω'),

which is s $(24)+(25)+$Alg. 4.3-1

Therefore, for every entry f in ω, there is an entry g in ω' whose value

is the same and whose transfer has the same source $[(21)+(22)+(23)+(26)+$

$(27)]$, so ω' is SOE-inclusive of ω $[(5)+(6)+(9)+(10)+(11)+(20)+$Def. 5.2-8].

$\triangle$

Theorem 7.1-4   EE(L_D,M) is a Structure-as-Storage model.

Proof:

(1) EE(L_D,M) = (V, L, A, In, E) is an entry-execution model [Thm. 4.3-1].

   There is a distinct subset V_p of V containing pointers [Defs. 2.2-1+

   4.3-1]. The action domain A contains the eight actions, and In

   assigns the input arities: Fetch (1), First (1), Next (2), Select

   (2), Copy (1), Assign (2), Update (3), and Delete (2) [Defs. 3.3-12+

   2.2-3+4.3-1].

(2) Let (Int,J) be any expansion in E, where Int = (St, I,IE). Then there

   is an L_D program P such that this pair is an expansion of P

                                                          Def. 4.3-1

(3) Let J be any job in J. Then J is a job for Int $[(1)+(2)+$Def. 4.2-3].

   Int = Int(P), and there is an equivalence class E of initial

   modified states for P such that J = J_E $[(2)+$Def. 4.3-2].

(4) P is an L_{BS} program $(2)+$Def. 3.3-12

(5) Let S_1 and S_2 be any two initial states in E, and let Ω_1 and Ω_2 be

any two halted firing sequences starting in S_1 and S_2. For $i=1,2$,

let S_i' be the initial standard state for P corresponding to S_i.

Then there is a halted firing sequence Ω_i' starting in S_i' such that

$\eta(S_i',\Omega_i')$ is SOE-inclusive of $\eta(S_i,\Omega_i)$ [(4)+Thm. 7.1-3]. Let

$\omega_i = \eta(S_i,\Omega_i)$ and $\omega_i' = \eta(S_i',\Omega_i')$

(6) ω_i' and ω_i are both causal computations for Int(P)   (5)+Lemma 4.3-2

(7) ω_i' satisfies the Input/Output Type Constraint [(5)+Lemma 5.3-1], the

   Structure Output Constraint [(5)+Lemma 5.3-3], and the Unique

   Pointer Generation Constraint [(5)+Lemma 5.3-6], all given Int(P)

For any pointer p which is the value in ω_i' of the output entries of a

Copy execution C, the first entry in ω_i' with value p is one of those

output entries of C [(6)+(7)+Lemma 5.3-8], so

(8) For any structure operation execution e initiated in ω_i and for any

   Assign, Update, or Delete execution A, $e \in R(A)$ in $\omega_i \rightarrow e \in R(A)$ in ω_i'

                                                                 (6)+(5)+Lemma 5.2-6

(9) ω_i satisfies the Input/Output Type, Structure Output, and Unique

   Pointer Generation Constraints, given Int(P)(5)+(6)+(7)+Lemma 5.3-9

(10) For any pointer p which is the value in ω_i of the output entries of

   a Copy execution C, the first entry in ω_i with value p is one of

   those output entries of C   (6)+(9)+Lemma 5.3-8

(11) Let β_i be any computation in J_{S_i,Ω_i}. Then β_i is causal [Def. 4.3-5],

   and β_i is in J_E, so it is a computation for Int = Int(P) [(3)+

   Defs. 4.3-4+4.2-3]

(12) ω_i is SOE-inclusive of β_i (5)+(4)+Lemma 5.3-7

(13) For any structure operation execution e initiated in β_i and any

   Assign, Update, or Delete execution A, $e \in R(A)$ in $\beta_i \rightarrow e \in R(A)$ in ω_i

[(6)+(11)+(12)+(10)+Lemma 5.2-6] $\wedge$ e is initiated in ω_i [(12)+

Def. 5.2-8] $\Rightarrow$ e$\in$R(A) in ω_i' [(8)]

(14) β_i satisfies the Input/Output Type, Structure Output, and Unique

Pointer Generation Constraints, given Int(P)

$$(6)+(11)+(12)+(10)+\text{Lemma } 5.3-9$$

(15) For any pointer p, p is the value of the output entries in β_i of a

Copy execution C $\Rightarrow$ the first entry in β_i with value p is one of

those output entries of C   (11)+(14)+Lemma 5.3-8

(16) Let α_i be any prefix of β_i. Let γf be any prefix of α_i and let e

be the execution of which f is an output entry. Then γf is a

prefix of β_i, so e is initiated in γ [(11)+Def. 4.2-7]. I.e.,

(17) α_i is causal

(18) α_i is in J, and so α_i, β_i, and ω_i are all computations for Int(P)

$$(16)+(11)+(6)+\text{Defs. } 4.3-3+4.2-3$$

(19) For any structure operation execution e = Ex(d,k) initiated in α_i

and any Assign, Update, or Delete execution A, e$\in$R(A) in β_i iff

e$\in$R(A) in α_i only if A is initiated in α_i [(17)+(11)+(18)+(16)+

(15)+Lemma 5.2-6] $\wedge$ there are In(I(d)) input entries to e in α_i,

hence in β_i, so e is initiated in β_i [Def. 4.2-6]

(20) e$\in$R(A) in ω_i' iff e$\in$R(A) in α_i only if A is initiated in α_i (19)+(13)

Let f be any entry in α_i. Then f is in β_i [(16)]. Let T(f) be

(Src(Ex(d,k),j), Dst(Ex(d',k'),j)). Then Constraint 5.1-1 dictates, one

or two times, what the type of V(f) should be, once based on I(d) and i,

and again based on I(d') and j. Both of the types so dictated match the

type of V(f) (since f is in β_i [(14)]). Therefore,

(21) α_i satisfies the Input/Output Type Constraint

(22) J satisfies the Pointer Transparency Constraint   Lemma 7.1-1

(23) Let e be any structure operation execution. If e is initiated in α_1, the e is initiated in β_1 [(19)], hence in ω_1 [(12)+Def. 5.2-8]. For every integer j, there is an entry $\text{Ent}_{\alpha_1}(e,j)$ in α_1 $\Rightarrow$ there is an entry $\text{Ent}_{\beta_1}(e,j)$ in β_1 with the same value [(16)] $\Rightarrow$ there is an entry $\text{Ent}_{\omega_1}(e,j)$ in ω_1 with the same value [(12)+Def. 5.2-8] $\Rightarrow$ there is an entry $\text{Ent}_{\omega_1'}(e,j)$ in ω_1' with the same value [(5)+Def. 5.2-8]. There is an entry in α_1 whose transfer has source $\text{Src}(e,j)$ only if there is an entry in ω_1 with the same value whose transfer has source $\text{Src}(e,j)$ [(16)+(12)+Def. 5.2-8] only if there is an entry in ω_1' with the same value whose transfer has source $\text{Src}(e,j)$ [(5)+ Def. 5.2-8]

(24) For any two pointers p_1 and p_2, $(p_1,\alpha_1)\rho(p_2,\alpha_2)$ $\Rightarrow$ $(p_1,\beta_1)\rho(p_2,\beta_2)$ [(17)+(11)+(18)+(16)+(19)+Lemma 5.3.10] $\Rightarrow$ $(p_1,\omega_1)\rho(p_2,\omega_2)$ [(11)+ (6)+(18)+(12)+(20)+Lemma 5.3-10] $\Rightarrow$ $(p_1,\omega_1')\rho(p_2,\omega_2')$ [(6)+(5)+(8)+ Lemma 5.3-10]

Since S_1' and S_2' are equal initial standard states for L_{BS} program P [(4)+(5)+Thm. 7.1-2], α_1 satisfies the Atomic Output, Structure Output, and Unique Pointer Generation Constraints, and α_1 and α_2 as a pair satisfies the Initial Structure and First/Next Output Constraints [(5)+(6)+(17)+(23)+(24)+Lemma 5.3-11]. Therefore, every computation in J_E satisfies the Input/Output Type, Atomic Output, Structure Output, and Unique Pointer Generation Constraints, every pair of computations satisfies the Initial Structure and First/Next Output Constraints, and J_E satisfies the Pointer Transparency Constraint [(16)+(11)+(5)+(21)+(22)+Def. 4.3-3].

From this and (1)-(3), $EE(L_D,M)$ is a Structure-as-Storage model [Ref. 5.1-[Def. 5.1-1].

<div align="right">Q.E.D.</div>

Lemma 7.2-3 For any equivalence class E of initial modified states for an L_{BS} program P, let J be J_E. Let Int(P) be (St,l,IE). Assume there are two computations αgf and $\alpha \overline{\overline{f}}\overline{g}$ in J such that $T(\overline{f}) = T(f)$, $T(\overline{g}) = T(g)$, and f and g initiate distinct executions $e_1 = Ex(d_1,k_1)$ and $e_2 = Ex(d_2,k_2)$ in αgf, where d_1 and d_2 are in St-DL. Let S and Ω (S' and Ω') be the state in E and halted firing sequence starting in that state such that αgf ($\alpha \overline{\overline{f}}\overline{g}$) is a prefix of a computation in $J_{S,\Omega}$ ($J_{S',\Omega'}$). Then there are prefixes $\theta \varphi_2 \varphi_1$ of Ω and $\theta' \varphi_1' \varphi_2'$ of Ω', whose reductions are $\Phi(\alpha gf)$ and $\Phi(\alpha \overline{\overline{f}}\overline{g})$, such that θ' equals θ and for i=1,2, φ_i (φ_i') is the k_i^{th} firing of d_i. Furthermore, φ_1 and φ_2 potentially interfere in $\theta \varphi_2 \varphi_1$ iff $Ent(e_1,1)$ and $Ent(e_2,1)$ are in the same access history, and e_1 is in $R(e_2)$, in αgf.

Proof: There is an expansion (Int,J) in $EE(L_{BS},M)$ such that Int = Int(P) and J$\in$$J$ [Defs. 4.3-2+4.3-1]. Hence, J = J_E is a job for Int(P) [Thm. 4.3-1+Def. 4.2-2], so

(1) αgf and $\alpha \overline{\overline{f}}\overline{g}$ are computations for Int(P)   Def. 4.2-3

(2) Let β (β') be the computation in $J_{S,\Omega}$ ($J_{S',\Omega'}$) of which αgf ($\alpha \overline{\overline{f}}\overline{g}$) is a prefix. Then, for i=1,2, the initiating entry to e_i in β_i (β_i') is preceded therein by the initiating entries to exactly k-1 other executions of d_i Cor. 4.3-1

(3) $\Phi(\alpha gf)$ equals $\Phi(\alpha)\overline{\varphi}_2\overline{\omega}_1$, in which $\overline{\varphi}_i$ is the k_i^{th} firing of d_i (2)+Def. 4.3-4

The prefix Δ of Ω whose length is the length of $\Phi(\alpha gf)$ has as its

reduction $\Phi(\alpha gf)$ [Lemma 7.2-2], so Δ can be written as $\theta \varphi_2 \varphi_1$, where θ is a prefix of Ω and φ_1 is the k_1^{th} firing of d_1 in Ω [(3)+Def. 2.4-5]. Then θ is the prefix of Ω whose length is two less than the length of Δ, so the length of θ is two less than the length of $\Phi(\alpha gf)$. I.e., θ is the prefix of Ω whose length is the length of $\Phi(\alpha)$ [(3)], so the reduction of θ is $\Phi(\alpha)$ [Lemma 7.2-2].

For $i=1,2$, there are $In(I(d_i))-1$ input entries to e_i in α [(1)+ Def. 4.2-6], and $\bar{f}$ and $\bar{g}$ are input entries to e_1 and e_2 respectively [Defs. 4.2-6+4.2-5], so $\bar{f}$ and $\bar{g}$ are the initiating entries to e_1 and e_2, respectively, in $\alpha \bar{f} \bar{g}$ [(1)+Def. 4.2-6]. The reasoning of the above paragraph applies, to give that there is a prefix $\theta' \varphi_1' \varphi_2'$ of Ω' whose reduction is $\Phi(\alpha \bar{f} \bar{g})$ such that $\Phi(\alpha)$ is the reduction of θ' and φ_i' is the k_i^{th} firing of d_i in Ω'. Since θ' and θ have the same reduction $\Phi(\alpha)$, they are equal [Def. 2.4-5].

(4) β is a permutation of $\omega = \eta(\mathcal{S},\Omega)$ [(2)+Def. 4.3-5], which is a computation for $Int(P)$ [Lemma 4.3-2]

(5) For any j and for $i=1,2$, the value of $Ent(e_i,j)$ in ω equals the value of the token removed from d_i's number-j input arc at d_i's k_i^{th} firing in Ω Alg. 4.3-1

(6) For $i=1,2$, all $In(I(d_i))$ input entries to e_i in ω are in αgf

        (1)+(2)+(4)+Def. 4.2-6

(7) $Ent(e_1,1)$ and $Ent(e_2,1)$ are in the same access history in αgf iff they have the same pointer value [Def. 5.1-4] iff $Ent(e_1,1)$ and $Ent(e_2,1)$ have the same pointer value in ω [(6)] iff φ_1 and φ_2 remove tokens with the same pointer value from the number-1 input arcs of d_1 and d_2 in Ω [(5)]

(8) There are entries $Ent(e_1,2)$ and $Ent(e_2,2)$ in αgf and their values are equal iff there are entries $Ent(e_1,2)$ and $Ent(e_2,2)$ in ω and their values are equal [(6)] iff φ_1 and φ_2 remove tokens of equal value from the number-2 input arcs of d_1 and d_2 in Ω [(5)]

(9) No execution initiates between e_1 and e_2 in αgf Def. 4.2-6

φ_1 and φ_2 potentially interfere in Ω iff they have equal number-1 inputs and either φ_2 is an Assign firing and φ_1 is a Fetch, Assign, or Copy firing, or φ_2 is an Update or Delete firing and φ_1 is a Copy, First, or Next firing or a Select, Update, or Delete firing with the same number-2 input as φ_2 [Def. 3.1-2] iff $Ent(e_1,1)$ follows $Ent(e_2,1)$ in the same access history in αgf, with no intervening entries [(7)+(9)+Def. 5.1-4] and either e_2 is an Assign execution and e_1 is a Fetch, Assign, or Copy execution, or e_2 is an Update or Delete execution and e_1 is a Copy, First, or Next execution, or a Select, Update, or Delete execution with $V(Ent(e_1,2)) = V(Ent(e_2,2))$ [(8)+Alg. 4.3-1] iff $Ent(e_1,1)$ and $Ent(e_2,1)$ are in the same access history in αgf and e_1 is in $R(e_2)$ in αgf [(9)+Defs. 5.1-5-5.1-8].

$\triangle$

Lemma 7.2-5 Let S_1 and S_2 be any two equal initial modified states for the same program P. Let Ω_1 and Ω_2 be two firing sequences starting in S_1 and S_2 respectively such that

(1) for each actor d in P, there are the same number of firings of d in both Ω_1 and Ω_2,

(2) for each gate d in P and each k, the kth firings of d in Ω_1 and Ω_2 remove control tokens of the same value, and

(3) for any two actors d and d', and for any k, there is a k' such that if the kth firings of d in Ω_1 and Ω_2 remove tokens from output arcs of d', then those firings both are preceded by k' firings of d'.

Then for any arc in P which holds tokens of pointer value in $S_1 \cdot \Omega_1$ and $S_2 \cdot \Omega_2$, either both are read pointers or both are write pointers.

Proof:

(4) Every token which appears on a program input arc has a read pointer as value   Def. 3.3-5

(5) Every token which appears on a number-1 output arc of a Copy has a write-pointer value, and every token which appears on the number-2 output arc of a Copy or the number-1 output arc of a Select has a read-pointer value   Def. 3.3-9

(6) Every arc can hold a token of pointer value only if it is a program input arc or an output arc of a Copy, Select, or pI actor

           Defs. 3.3-9+2.2-5

Prove by contradiction that for every pI actor d in P and every integer $k > 0$, the kth firing of d in Ω_1 outputs a read (write) pointer iff the kth firing of d in Ω_2 outputs a read (write) pointer. Assume

(7) the above is false

(8) There is a prefix $\theta\varphi$ of Ω_1 such that for every pI actor d' and integer k' such that there are no more than k' firings of d' in θ, the k'^{th} firing of d' in Ω_1 outputs a read (write) pointer iff the k'^{th} firing of d' in Ω_2 outputs a read (write) pointer, and φ is the kth firing of pI actor d, it outputs a read (write) pointer in Ω_1, and the kth firing of d in Ω_2 does not   (7)

By (2), d is a gate $\Rightarrow$ φ removes a <u>true</u> control token iff the kth firing of d in Ω_2 removes a <u>true</u> control token. Hence,

(9) there is one input arc b of d such that a token is copied from b to d's output arcs by the kth firing of d in both Ω_1 and Ω_2 [Defs. 2.1-5+2.2-4]. b is a program input arc or an output arc of a Copy or Select $\Rightarrow$ the kth firings of d in Ω_1 and Ω_2 either both output read pointers or both output write pointers [(4)+(5)], so b is an output arc of a pI actor d' [(8)+(6)]

(10) The tokens output by φ in Ω_1 are identical to those output by the k'^{th} firing of d', where there are exactly k' firings of d' in θ [(9)+Def. 2.1-5], so there are exactly k' firings of d' before the kth firing of d in Ω_2 [(9)+(3)]. Hence the tokens output by the kth firing of d in Ω_2 are identical to those output by the k'^{th} firing of d' in Ω_2 [(9)+Def. 2.1-5].

The tokens output by φ are read (write) pointers iff the tokens output by the kth firing of d in Ω_2 are read (write) pointers [(10)+(8)]. Since (7) leads to this contradiction with (8), (7) is false; i.e.,

(11) for every pI actor d and integer k > 0, the kth firing of d in Ω_1 outputs a read (write) pointer iff the kth firing of d in Ω_2 outputs a read (write) pointer

Letting b be any arc which holds a token of value (p,R) or (p,W) in both $S_2 \cdot \Omega_2$ and $S_1 \cdot \Omega_1$, b is an output arc of a pI actor d $\Rightarrow$ the token on b in $S_1 \cdot \Omega_1$ was output by the kth firing of d in Ω_1, where k is the exact number of firings of d in Ω_1 [Def. 2.1-5] $\Rightarrow$ there are exactly k firings of d in Ω_2 [(1)] $\Rightarrow$ the token on b in $S_2 \cdot \Omega_2$ was placed there by the kth firing of d in Ω_2 [Def. 2.1-5]. Therefore, the token on b in $S_1 \cdot \Omega_1$ is a read

(write) pointer iff the token on b in $S_2 \cdot \Omega_2$ is a read (write) pointer [(6)+(4)+(5)+(11)].

$\triangle$

Lemma 7.2-6 Given any L_D program P, let Ω be any halted firing sequence starting in any initial modified state S for P. Let $\theta\varphi_2\varphi_1$ be any prefix of Ω and let Ξ be such that $\Omega = \theta\varphi_2\varphi_1\Xi$. If $\theta\varphi_1\varphi_2$ is a firing sequence starting in S and $S \cdot \theta\varphi_1\varphi_2$ is identical to $S \cdot \theta\varphi_2\varphi_1$, then $\Omega' = \theta\varphi_1\varphi_2\Xi$ is a halted firing sequence starting in S and $\eta(S,\Omega')$ contains the same set of entries as $\eta(S,\Omega)$.

Proof:

Key definitions: Def. 2.3-1 - firing sequence starting in a state; Defs. 3.3-9+3.3-7+2.1-5 - modified interpreter; Alg. 4.3-1 - $\omega(S,\Omega)$

First prove the following hypotheses by induction on the lengths of the prefixes Δ of Ξ:

  A: $\theta\varphi_1\varphi_2\Delta$ is a firing sequence starting in S

  B: $S \cdot \theta\varphi_1\varphi_2\Delta$ is identical to $S \cdot \theta\varphi_2\varphi_1\Delta$

  C: $\omega(S,\theta\varphi_1\varphi_2\Delta)$ contains the same set of entries as $\omega(S,\theta\varphi_2\varphi_1\Delta)$

Basis: $|\Delta| = 0$.

A and B are true by Lemma hypothesis.

  (1) Let d_1 and d_2 be the actors in P of which φ_1 and φ_2 are firings.
     Then both are enabled in $S \cdot \theta$

If either is a gate, its control input arc has a token on it in $S \cdot \theta$, and so that arc is not an output arc of the other actor [(1)+Defs. 3.3-6+ 2.1-4]. If d_1 (d_2) is a gate, then the control token input by φ_1 (φ_2)

in either $\theta\varphi_1\varphi_2$ or $\theta\varphi_2\varphi_1$ is the token on its control input arc in $S\cdot\theta$. Therefore,

(2) The set of input arcs from which φ_1 (φ_2) removes tokens is the same in both $\theta\varphi_1\varphi_2$ and $\theta\varphi_2\varphi_1$. All of those arcs have tokens on them in $S\cdot\theta$ [(1)+Defs. 3.3-6+2.1-4], so none of those arcs is an output arc of either d_1 or d_2 [(1)+Defs. 3.3-6+2.1-4].

(3) All of the tokens removed by φ_1 (φ_2) in either $\theta\varphi_1\varphi_2$ or $\theta\varphi_2\varphi_1$ are on the arcs from which they are removed in $S\cdot\theta$ (2)

(4) Let b be any input arc of d_1 (d_2) from which a token is removed by φ_1 (φ_2) in either $\theta\varphi_1\varphi_2$ or $\theta\varphi_2\varphi_1$. Then there is a token on b in $S\cdot\theta\varphi_2$ and $S\cdot\theta$ ($S\cdot\theta\varphi_1$ and $S\cdot\theta$)   (3)

If b is an output arc of actor d, then there are the same number of firings of d in $\theta\varphi_1$ and $\theta\varphi_2$ as in θ [(2)], so

(5) Source(b,S,$\theta\varphi_1$) = Source(b,S,θ) (Source(b,S,$\theta\varphi_2$) = Source(b,S,θ))

                                                  (4)+Lemma 7.1-3

(6) All of the entries in $\omega(S,\theta)$ are in each of $\omega(S,\theta\varphi_1\varphi_2)$ and $\omega(S,\theta\varphi_2\varphi_1)$

(7) There is an entry with value v and transfer (s, Dst(Ex(d,k),j)) in $\omega(S,\theta\varphi_1\varphi_2)$ that is not in $\omega(S,\theta)$ iff $d = d_1$, φ_1 is the kth firing of d_1 in $\theta\varphi_1\varphi_2$, φ_1 removes a token of value v from b, the number-j input arc of d_1, and s = Source(b,S,θ), or $d = d_2$, φ_2 is the kth firing of d_2 in $\theta\varphi_1\varphi_2$, φ_2 removes a token of value v from b, d_2's number-j input arc, and s = Source(b,S,$\theta\varphi_1$) iff $d = d_1$, φ_1 is the kth firing of d_1 in $\theta\varphi_2\varphi_1$, φ_1 removes a token of value v from b, the number-j input arc of d_1, in $\theta\varphi_2\varphi_1$, and s = Source(b,S,$\theta\varphi_2$) [(2)+(3)+(5)], or $d = d_2$, φ_2 is the kth firing of d_2 in $\theta\varphi_2\varphi_1$, φ_2 removes a token of value v from b, the number-j input arc of d_2,

and $s = \text{Source}(b,S,\theta)$ $[(2)+(3)+(5)]$ iff there is an entry with value v and transfer $(s, \text{Dst}(\text{Ex}(d,k),j))$ in $\omega(S,\theta\varphi_2\varphi_1)$ which is not in $\omega(S,\theta)$

Therefore, $\omega(S,\theta\varphi_1\varphi_2)$ contains the same set of entries as $\omega(S,\theta\varphi_2\varphi_1)$ $[(6)+(7)]$.

Induction step: Assume that A, B, and C are true for prefix Δ of Ξ. $0 \le |\Delta| < |\Xi|$, and consider prefix $\Delta\varphi$ of Ξ.

(8) $\theta\varphi_2\varphi_1\Delta\varphi$ is a firing sequence starting in S

(9) Let d be the actor of which φ is a firing. Then d is enabled in

$$S \cdot \theta\varphi_2\varphi_1\Delta \qquad\qquad (8)$$

(10) Enabling conditions for an actor are a function solely of state

Defs. 3.3-6+2.1-4

(11) $S \cdot \theta\varphi_1\varphi_2\Delta$ is identical to $S \cdot \theta\varphi_2\varphi_1\Delta$ ind. hyp. B

d is enabled in $S \cdot \theta\varphi_1\varphi_2\Delta$ $[(8)+(10)+(9)]$, and d is a Copy and $\varphi = (d,(p,n))$ $\Rightarrow$ (p,n) is not in Π in $S \cdot \theta\varphi_2\varphi_1\Delta$ $[(8)+\text{Defs. } 2.3-1+2.2-5]$ $\Rightarrow$ (p,n) is not in Π in $S \cdot \theta\varphi_1\varphi_2\Delta$ $[(11)]$ $\Rightarrow$ (p,n) can be added to Π in going to $S \cdot \theta\varphi_1\varphi_2\Delta\varphi$ [Def. 2.2-5]. Therefore, $\theta\varphi_1\varphi_2\Delta\varphi$ is a firing sequence starting in S [ind. hyp. A+Def. 2.3-1].

The state after a state transition depends only on the state before the transition, the actor chosen to fire, and if that is a Copy, the pair (p,n) chosen to be added to Π. Therefore,

(12) $S \cdot \theta\varphi_1\varphi_2\Delta\varphi$ is identical to $S \cdot \theta\varphi_2\varphi_1\Delta\varphi$

(13) If d is a gate, then φ has the same control input in both $\theta\varphi_1\varphi_2\Delta\varphi$ and $\theta\varphi_2\varphi_1\Delta\varphi$ (11)

(14) $\omega(S,\theta\varphi_2\varphi_1\Delta)$ is a prefix of $\omega(S,\theta\varphi_2\varphi_1\Delta\varphi)$ and $\omega(S,\theta\varphi_1\varphi_2\Delta)$ is a prefix of $\omega(S,\theta\varphi_1\varphi_2\Delta\varphi)$

(15) Let b be any arc from which φ removes a token in either $\theta\varphi_2\varphi_1\Delta\varphi$ or $\theta\varphi_1\varphi_2\Delta\varphi$. Then there is a token on b in both $S \cdot \theta\varphi_2\varphi_1\Delta$ and $S \cdot \theta\varphi_1\varphi_2\Delta$. [(12)]. If b is an output arc of an actor, then there are the same number of firings of that actor in $\theta\varphi_2\varphi_1\Delta$ and $\theta\varphi_1\varphi_2\Delta$, so Source$(b,S,\theta\varphi_2\varphi_1\Delta) =$ Source$(b,S,\theta\varphi_1\varphi_2\Delta)$ [Lemma 7.1-3]

(16) There is an entry with value v and transfer (s, Dst(Ex(d,k),j)) in $\omega(S,\theta\varphi_1\varphi_2\Delta\varphi)$ which is not in $\omega(S,\theta\varphi_1\varphi_2\Delta)$ iff φ is the kth firing of d in $\theta\varphi_1\varphi_2\Delta\varphi$, it removes a token of value v from b, the number-j input arc of d, in $\theta\varphi_1\varphi_2\Delta\varphi$, and s = Source$(b,S,\theta\varphi_1\varphi_2\Delta)$ iff φ is the kth firing of d in $\theta\varphi_2\varphi_1\Delta\varphi$, it removes a token of value v from b, the number-j input arc of d, in $\theta\varphi_2\varphi_1\Delta\varphi$ [(13)+(11)] and s is Source$(b,S,\theta\varphi_2\varphi_1\Delta)$ [(15)] iff there is an entry with value v and transfer (s, Dst(Ex(d,k),j)) in $\omega(S,\theta\varphi_2\varphi_1\Delta\varphi)$ which is not in $\omega(S,\theta\varphi_2\varphi_1\Delta)$

Therefore, $\omega(S,\theta\varphi_2\varphi_1\Delta\varphi)$ contains the same set of entries as $\omega(S,\theta\varphi_1\varphi_2\Delta\varphi)$ [(14)+(16)]. Thus it is proven by induction that

(17a) Ω' is a firing sequence starting in S

(17b) $S \cdot \Omega'$ and $S \cdot \Omega$ are identical states

(17c) $\omega(S,\Omega')$ contains the same set of entries as $\omega(S,\Omega)$

Since Ω is halted, there is no actor enabled in $S \cdot \Omega$ [Def. 2.3-1], so there is no actor enabled in $S \cdot \Omega'$ [(10)+(17b)], so Ω' is a halted firing sequence starting in S [(17a)].

(18) $\omega(S,\Omega)$ is a prefix of $\eta(S,\Omega)$ and $\omega(S,\Omega')$ is a prefix of $\eta(S,\Omega')$

<div align="right">Alg. 4.3-1</div>

Let b be any arc which holds a token in $S \cdot \Omega$ or $S \cdot \Omega'$. Then b holds a token in both $S \cdot \Omega$ and $S \cdot \Omega'$ [(17b)]. If b is an output arc of an actor d, there

are the same number of firings of d in $\Omega = \theta\varphi_2\varphi_1\Xi$ as in $\Omega' = \theta\varphi_1\varphi_2\Xi$, so

Source(b,S,Ω) = Source(b,S,Ω') [Lemma 7.1-3]. Therefore,

(19) there is an entry with value v and transfer (s, d) in $\eta(S,\Omega')$ which

is not in $\omega(S,\Omega')$ iff there is an arc b which holds a token of value

v in $S\cdot\Omega'$, s = Source(b,S,Ω'), and d is a certain fixed function

of b [Alg. 4.3-1] iff there is an arc b which holds a token of

value v in $S\cdot\Omega$ [(17b)], s = Source(b,S,Ω) and d is a certain

fixed function of b iff there is an entry with value v and transfer

(s, d) in $\eta(S,\Omega)$ which is not in $\omega(S,\Omega)$ [Alg. 4.3-1]

Thus, $\eta(S,\Omega')$ contains the same set of entries as $\eta(S,\Omega)$ [(18)+(19)].

$\triangle$

Bibliography

[1]   Ackerman, W.B.   "A Structure Processing Facility for Data Flow
            Computers." *Proceedings of the 1978 International Conference
            on Parallel Processing.* Wayne State University, Aug. 1978,
            pp. 168-172.

[2]   Adams, D.A.   "A Model for Parallel Computations." In *Parallel
            Processor Systems, Technologies, and Applications.*
            Ed. L. C. Hobbs, et al. New York: Spartan Books, 1970,
            pp. 311-334.

[3]   Arvind and K.P. Gostelow. *A New Interpreter for Data Flow Schemas
            and Its Implications for Computer Architecture.* Technical
            Report #72, Dept. of Information and Computer Science,
            University of California, Irvine, Oct. 1975.

[4]   Arvind and K.P. Gostelow. "A Computer Capable of Exchanging
            Processors for Time." *Information Processing 77.* New York:
            North-Holland, 1977, pp. 849-854.

[5]   Ashcroft, E.A. "Proving Assertions about Parallel Programs."
            Journal of Computer and System Science, vol. 10, no. 1
            (Jan. 1975), pp. 110-135.

[6]   Campbell-Grant, I. *The Controlled Execution of Parallel Programs
            on Structured Data.* S.M. Thesis, Dept. of Electrical
            Engineering, M.I.T., January 1971.

[7]   Codd, E.F. "A Relational Model of Data for Large Shared Data Banks."
            Communications of the ACM, vol. 13, no. 6 (June 1970),
            pp. 377-397.

[8]   Davis, A.L. "The Architecture and System Method of DDM1:
            A Recursively Structured Data Driven Machine." *Proceedings of
            the Fifth Annual Symposium on Computer Architecture.* Palo Alto,
            Calif.,   April 1978, pp. 210-215.

[9]   Denning, P.J. "Virtual Memory." *Computing Surveys,* vol. 2, no. 3
            (Sept. 1970), pp. 153-189.

[10]   Denning, P.J. "On the Determinacy of Schemata." *Record of the
            Project MAC Conference on Concurrent Systems and Parallel
            Computation.* New York: ACM, 1970, pp. 143-147.

-562-

[11] Dennis, J.B. "Programming Generality, Parallelism, and Computer
        Architecture." Computation Structures Group Memo 32, M.I.T.
        Laboratory for Computer Science, August 1968. Also in
        Information Processing 68. Amsterdam: North-Holland, 1969,
        pp. 484-492.

[12] Dennis, J.B. "First Version of a Data Flow Procedure Language."
        Computation Structures Group Memo 93-1, M.I.T. Laboratory for
        Computer Science, August 1974.

[13] Dennis, J.B. and J.B. Fosseen. "Introduction to Data Flow Schemas."
        Computation Structures Group Memo 81, M.I.T. Laboratory for
        Computer Science, June 1973.

[14] Dennis, J.B. and D.P. Misunas. "A Preliminary Architecture for a
        Basic Data-Flow Processor." *Proceedings of the Second
        Symposium on Computer Architecture*. University of Houston,
        1975, pp. 126-132.

[15] Dennis, J.B., D.P. Misunas, and C.K. Leung. "A Highly Parallel
        Processor Using a Data Flow Machine Language." Computation
        Structures Group Memo 134, M.I.T Laboratory for Computer
        Science, January 1977.

[16] Deutsch, L.P. and D.G. Bobrow. "An Efficient, Incremental,
        Automatic Garbage Collector." *Communications of the ACM*,
        vol. 19, no. 9 (Sept. 1976), pp. 522-526.

[17] Fox, P.S. *Representation of Parallel Computation on Data Structures*.
        S.M. and E.E. Thesis, Dept. of Electrical Engineering, M.I.T.,
        January 1973.

[18] Gertz, J.L. *Hierarchical Associative Memories for Parallel
        Computation*. Ph.D. Thesis, Dept. of Electrical Engineering,
        M.I.T., June 1970. Also MAC-TR-69, M.I T. Laboratory for
        Computer Science.

[19] Greif, I. *Semantics of Communicating Parallel Processes*. Ph.D.
        Thesis, Dept. of Electrical Engineering and Computer Science,
        M.I.T., Aug. 1975. Also MAC-TR-154, M.I.T. Laboratory for
        Computer Science.

[20] Gurd, J. and I. Watson. "A Multilayered Data Flow Computer
        Architecture." *Proceedings of the 1977 International Conference
        on Parallel Processing*. Wayne State University, Aug. 1977,
        p. 94.

      Hewskiewycz, I.T. *Semantics of Data Base Systems*. Ph.D. Thesis,
        Dept. of Electrical Engineering, M.I.T., June 1973. Also
        MAC-TR-112, M.I.T. Laboratory for Computer Science.

[22] Kahn, G. "The Semantica of a Simple Language for Parallel
      Programming." *Information Processing 74*. Amsterdam:
      North-Holland, 1974, pp. 471-475.

[23] Karp, R.M. and R.E. Miller. "Properties of a Model for Parallel
      Computations: Determinacy, Termination, Queuing." *SIAM
      Journal of Applied Mathematics*, vol. 14, no. 6 (Nov. 1966),
      pp. 1390-1411.

[24] Karp, R.M. and R.E. Miller. "Parallel Program Schemata." *Journal
      of Computer and System Science*, vol. 3, no. 4 (May 1969),
      pp. 167-195.

[25] Knuth, D.E. *Fundamental Algorithms*, 2nd ed. Vol. 1 of *The Art
      of Computer Programming*. Reading, Mass.: Addison-Wesley, 1973.

[26] Korn, G.A. *Minicomputers for Engineers and Scientists*. New York:
      McGraw-Hill, 1973.

[27] Kosinski, P. "Mathematical Semantics and Data Flow Programming."
      *Proceedings of the 3rd ACM Symposium on Principles of
      Programming Languages*. Atlanta, 1976, pp. 175-184.

[28] Luconi, F.L. *Asynchronous Computational Structures*. Ph.D. Thesis,
      Dept. of Electrical Engineering, M.I.T., Feb. 1968. Also
      MAC-TR-49, M.I.T. Laboratory for Computer Science.

[29] Richards, H., Jr. and R.J. Zingg. "The Logical Structure of the
      Memory Resource in the SYMBOL-2R Computer." *Proceedings of the
      High Level Language Computer Architecture Symposium*. College
      Park, Md., 1973, pp. 1-10.

[30] Rodriguez, J.E. *A Graph Model for Parallel Computation*. Ph.D.
      Thesis, Dept. of Electrical Engineering, M.I.T., Sept. 1967.
      Also MAC-TR-64, M.I.T. Laboratory for Computer Science.

[31] Rumbaugh, J. *A Parallel Asynchronous Computer Architecture for
      Data Flow Programs*. Ph.D. Thesis, Dept. of Electrical
      Engineering and Computer Science, M.I.T., May 1975. Also
      MAC-TR-150, M.I.T. Laboratory for Computer Science.

[32] Rumbaugh, J. "A Data Flow Multiprocessor." *IEEE Transactions on
      Computers*, vol. C-26, no. 2 (Feb. 1977), pp. 138-146.

[33] Schroeder, M.A. and R.A. Meyer. "A Distributed Computer System
      Using a Data Flow Approach." *Proceedings of the 1977
      International Conference on Parallel Processing*. Wayne State
      University, Aug. 1977, p. 93.

[34] Slutz, D.R. *The Flow Graph Schemata Model of Parallel Computation.* Ph.D. Thesis, Dept. of Electrical Engineering, M.I.T., Sept. 1968. Also MAC-TR-53, M.I.T. Laboratory for Computer Science.

[35] Syre, J.C., D. Comte, and N. Hifdi. "Pipelining, Parallelism, and Asynchronism in the LAU System." *Proceedings of the 1977 International Conference on Parallel Processing.* Wayne State University, Aug. 1977, pp. 87-92.

[36] Thurber, K.J. and P.C. Patton, *Data Structures and Computer Architecture.* Lexington, Mass.: Lexington Books, 1977.

Biographical Note

David L. Isaman was born on August 15, 1947, in Endicott, New York.
He attended school in Petaluma and San Jose, California, graduating
from Leigh High School, San Jose, in 1964. In 1968, he received the
B.S. degree in Engineering (Electrical) from the California Institute
of Technology, graduating with honors. He received an NSF Graduate
Fellowship to attend the Massachusetts Institute of Technology, gaining
an M.S. degree in 1970. From 1975 to 1978, he was a member of the
computer science faculty at the University of California, San Diego.