CONGRUENT

WEAK CONFORMANCE

DISSERTATION

Ronald W. Brower, Civilian, USAF

AFIT/DS/ENG/02-04

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

Report Documentation Page

Report Date Sep 02	Report Type Final	Dates Covered (from... to) Dec 97 - Aug 02

Title and Subtitle Congruent Weak Conformance	**Contract Number**
	Grant Number
	Program Element Number
Author(s) Mr. Ronald Brower, DR-II, USAF	**Project Number**
	Task Number
	Work Unit Number
Performing Organization Name(s) and Address(es) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 P Street, Bldg 640 WPAFB OH 45433-7765	**Performing Organization Report Number** AFIT/DS/ENG/02-04
Sponsoring/Monitoring Agency Name(s) and Address(es)	**Sponsor/Monitor's Acronym(s)**
	Sponsor/Monitor's Report Number(s)

Distribution/Availability Statement
Approved for public release, distribution unlimited

Supplementary Notes

Abstract
This research addresses the problem of verifying implementations against specifications through an innovative logic approach. Congruent weak conformance, a formal relationship between agents and specifications, has been developed and proven to be a congruent partial order. This property arises from a set of relations called weak conformations. The largest, called weak conformance, is analogous to Milners observational equivalence. Weak conformance is not an equivalence, however, but rather an ordering relation among processes. Weak conformance allows behaviors in the implementation that are unreachable in the specification. Furthermore, it exploits output concurrencies and allows interleaving of extraneous output actions in the implementation. Finally, reasonable restrictions in CCS syntax strengthen weak conformance to a congruence, called congruent weak conformance. At present, congruent weak conformance is the best known formal relation for verifying implementations against specifications. This precongruence derives maximal flexibility and embodies all weaknesses in input, output, and no-connect signals while retaining a fully replaceable conformance to the specification. Congruent weak conformance has additional utility in verifying transformations between systems of incompatible semantics. This dissertation describes a hypothetical translator from the informal simulation semantics of VHDL to the bisimulation semantics of CCS. A second translator is described from VHDL to a broadcast-communication version of CCS. By showing that they preserve congruent weak conformance, both translators are verified.

Subject Terms
Asynchronous Systems, Automata, Bisimulation, CCS, Concurrency, Congruence, Digital Systems, Formal Methods, Precongruence, Preorder, Process Algebra, Semantics, Simulation, Specifications, Verification, VHDL.

Report Classification	Classification of this page
unclassified	unclassified
Classification of Abstract	**Limitation of Abstract**
unclassified	UU
Number of Pages	
199	

AFIT/DS/ENG/02-04

CONGRUENT WEAK CONFORMANCE

DISSERTATION

Presented to the Faculty

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

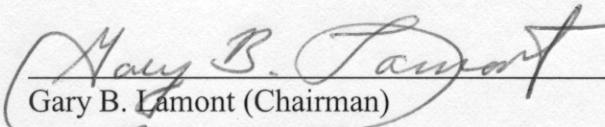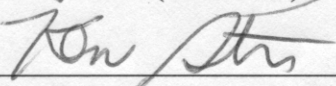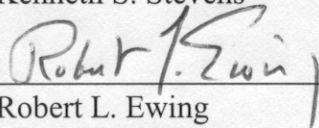Degree of Doctor of Philosophy
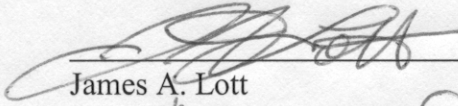
Ronald W. Brower, BS, MS
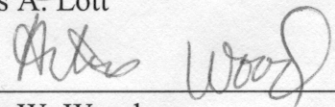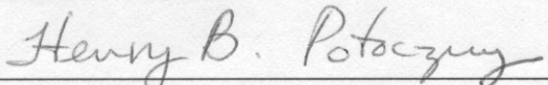
Civilian, USAF

September 2002

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.
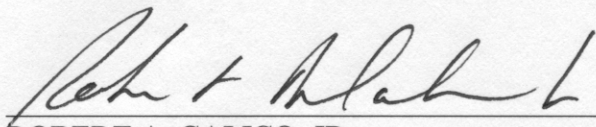
CONGRUENT WEAK CONFORMANCE

Ronald W. Brower, B.S., M.S.
Civilian, USAF

Approved:

	Date
Gary B. Lamont	5 - AUG - 02
Gary B. Lamont (Chairman)	
Ken Stevens	5 - AUG - 02
Kenneth S. Stevens	
Robert L. Ewing	5 - Aug - 02
Robert L. Ewing	
James A. Lott	05 August 2002
James A. Lott	
Aihua Wood	05 - Aug. - 02
Aihua W. Wood	
Henry B. Potoczny	5 Aug 2002
Henry B. Potoczny (Dean's Representative)	

Accepted:

Robert A. Calico

ROBERT A. CALICO, JR.
Dean, Graduate School of Engineering and Management

Table of Contents

List of Figures

List of Tables

List of Symbols

CCS

$\overset{def}{=}$	*Constant* operator (assignment)
.	*Prefix*
+	*Choice* or *Summation*
|	*Parallel Composition*, also concurrency of actions (informally)
|$_c$	*Parallel Conjunction* (*Broadcast* CCS only)
\	*Restriction*
[...]	*Relabeling*
a	action (lower case name)
$\overline{a}$, $'a$	coaction (overbarred or ticked lower case name)
τ	hidden or silent action
A	agent or process (Capitalized)
NIL	agent capable of no actions

Actions and Transitions

$\overset{a}{\rightarrow}$	literal transition
$\overset{s}{\Rightarrow}$	transition with τ abstracted.
$\Rightarrow$	equivalent to $(\overset{\tau}{\rightarrow})^*$
$\mathcal{A}ct$	atomic action set (all actions, coactions, and τ)
$\mathcal{A}$	input Action set
$\overline{\mathcal{A}}$	output Action set
$\mathcal{L}$	visible label set $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$
$\mathcal{A}(P), \mathcal{L}(P)$,etc.	corresponding set peculiar to agent P
$\mathcal{E}xtr(I,S)$	extraneous inputs of I over S
$\overline{\mathcal{E}xtr}\,(I,S)$	extraneous outputs of I over S
$\overline{\mathcal{I}dle}\,(I\ \mathcal{W}\ S)$	idle output actions of I with respect to S under $\mathcal{W}$.
$\hat{s}$	projection of s onto $\mathcal{L}$ (removes all τ *actions* from s)
ε	empty sequence
/	excess of...over...
$=_{conf}$	confluence equivalence
$\upharpoonright$	projection
τ_S	relative τ

Process Equivalences

$\sim_t$	trace equivalence
$\approx$	observational equivalence
$=$	observational congruence
$\sim$	strong equivalence

Ordering Relations

$\geq$ General ordering relation

$\succeq_l$ Logic conformance

$\mathscr{W}, \mathscr{V}$ weak conformation (capitalized Monotype Corsiva letters)

$\succeq_w$ Weak conformance

$\succeqq_w$ Congruent weak conformance

Modal and Temporal Logic

$\models$ satisfaction

$[a]$ necessity

$<a>$ possibility

$-$ ranges over entire action set

$\square$ Always

$\lozenge$ Possible

EV Eventually

$\bigcirc$ Next

$min(X.F(X))$ minimum fixed point of $X = F(X)$

$max(X.F(X))$ maximum fixed point of $X = F(X)$

Other

$\circ$ relational composition

Id_p process identity

AFIT/DS/ENG/02-04

Abstract

This research addresses the problem of verifying implementations against specifications through an innovative logic approach. *Congruent weak conformance*, a formal relationship between agents and specifications, has been developed and proven to be a congruent partial order. This property, symbolized $\succeq_w$, arises from a set of relations called *weak conformations*. The largest, called *weak conformance*, is analogous to Milner's *observational equivalence*. Unlike observational equivalence, however, weak conformance is not an equivalence, but rather an ordering relation among processes. Like the previous property of *logic conformance*, weak conformance allows behaviors in the implementation that are unreachable in the specification. Unlike logic conformance, however, weak conformance exploits output concurrencies and allows interleaving of extraneous output actions in the implementation. Finally, reasonable restrictions in design models strengthen weak conformance to a congruence. Being both congruent and a partial order, it merits the customary term *precongruence*. At this writing, $\succeq_w$ is the best known formal relation for verifying implementations against specifications. This precongruence derives maximal flexibility and embodies all weaknesses in input, output, and no-connect signals while retaining a fully replaceable conformance to the specification. This desirable relation is described in four transitional laws with five constructional restrictions.

Congruent weak conformance has additional utility in verifying transformations between systems of incompatible semantics such as found in circuit development,

security system design, and software engineering. This dissertation describes a hypothetical translator from the informal simulation semantics of VHDL to the bisimulation semantics of CCS. A second translator is described from VHDL to a broadcast-communication version of CCS. By showing that they preserve congruent weak conformance, both translators are verified.

CONGRUENT WEAK CONFORMANCE

I. Introduction

Engineers are continually challenged to produce electronic designs that meet specification; and logisticians are forever seeking replacements for obsolete, non-procurable microcircuits. Thus there is a general need to find circuits and circuit models that are "equivalent" either to a specification model or to some obsolete part that needs to be replaced. However a moment's reflection reveals that *equivalence* is a stronger notion than what is really needed or desired.

First of all, equivalent *speed* is not necessary. One can often replace an obsolete circuit with a faster circuit of equivalent functionality. This approach springs from the rationalization that the faster part can certainly keep pace with the system demands, while any tight timing constraints simply become less stringent. However, introducing a speedier component can uncover race conditions and hazards that were safeguarded by the delays inherent in the original component. In fact, practitioners often deliberately introduce delays to recover timing safeguards when faster parts are used.

Secondly, excess or redundant circuitry in the implementation can often be tolerated. The extra circuitry can simply sit idle, with pins either unconnected or grounded. Also, unneeded behaviors at connected pins can often be ignored during certain phases of the execution.

Thirdly, options allowed by output concurrency can be exploited. If the specification calls for the production of two *concurrent* outputs x and y, then both output

interleavings: *x* followed by *y,* and *y* followed by *x,* are admissible. The original implementing device may consistently produce one interleaving with the replacement device producing the other interleaving. One would never consider the two devices "equivalent," yet each may serve equally well within a specific application.

This dissertation introduces a new property called *congruent weak conformance* to capture the desired relationship between a specification and a compliant implementation. Congruent weak conformance is not a true equivalence, but rather a *partial order* that formally embodies intuitive notions of compliance. This precongruence is the least constraining formal relation known, as of this writing, for verifying implementations against specifications. It derives maximal flexibility and embodies all weaknesses in input, output, and no-connect signals while retaining a fully replaceable conformance to the specification. This desirable relation is described in four transitional laws with five constructional restrictions.

Congruent weak conformance also provides a link between formalisms of differing semantics. Whenever transformations are proposed for translating from one representation to another, congruent weak conformance must be preserved by such transformations, even when other semantic information is lost. In particular, transformations from the informal *simulation semantics* of hardware description languages such as VHDL (IEEE, 1993) to process algebras such as the *Calculus of Communicating Systems* (*CCS*) (Milner, 1989) can be validated, allowing stricter verifications of VHDL models based on the *bisimulation semantics* of CCS.

1.1 Background

The design of digital very-large-scale integrated (VLSI) circuits will be greatly aided if formal, high-level languages support all life cycle activities, including:

specification, simulation, synthesis, verification, documentation, testing, procurement, replacement and reengineering. Such languages provide the opportunity to automate many, if not all, of these activities. Such automation in turn decreases expense, shortens delivery time, and increases the likelihood that delivered parts meet the user's needs.

1.1.1 Language-based Life Cycle Activities. An example of such a high-level language is the *VHSIC Hardware Description Language* (VHDL), a standardized language managed by the Institute of Electrical and Electronic Engineers (IEEE). The United States Department of Defense (DoD) has, in the past, required VHDL for microcircuit documentation (DoD, 1992).

The semantics of VHDL is presented informally in the *VHDL Language Reference Manual* as *simulation* semantics (IEEE, 1993). Having such semantics, VHDL is widely used as the input language for the simulation of digital electronic components and systems. Designers use simulation to predict the behavior of their designs before implementation. During simulation, a *behavioral* VHDL model, which represents a set of requirements (specification), and a *structural* VHDL model, which represents a physical design, are compared by being subjected to the same set of input stimuli known as a VHDL *test bench*. Thus the language has an up-front role in the design process by expressing specification requirements and by aiding in the debugging of new designs. Another major use of VHDL is as the input language for the automatic *synthesis* of a design directly from a specification.

VHDL's support for microcircuit *testing* is solidified through ongoing work on the WAVES standard (IEEE, 1991). WAVES is a standardized subset of VHDL used for the description of test vectors. Adherence to the WAVES standard will assure that the same vectors used to check out a hardware design can be used to test the physical hardware as well.

Having supported other life cycle activities, if VHDL can now support formal verification as well, it will have a strengthened role as the *lingua franca* of electronic design.

1.1.2 Life Cycle Terminology. To alleviate confusion that may result from vague and overlapping usage of certain common terms that refer to various aspects of the integrated circuit life cycle, the following usage will apply in the ensuing discussion:

System. The term *system* refers to the end item, box, appliance, or printed circuit board that uses integrated circuits as components. The system design effort is a separate and higher level function than component design, yet intimate knowledge of component behavior is used in the system design process.

Component. A *component* is a single, monolithic integrated circuit. The terms *part* or *device* may also be used to designate a component.

Implementation. This is the physical realization of a component.

Design. The term *design*, when used alone, refers specifically to the design of a component and not the design of a system (which is explicitly called a *system design*). A *design* is thus some representation of an envisioned implementation for some component. Since an implementation can often be mechanically generated from a completed design, the terms *design* and *implementation* will often be interchangeable.

Designer. Similarly, a *designer* is specifically a component designer. A system designer will always be qualified with the word *system*.

Environment. For each of its components, the system provides an *environment*. The environment is the complete set of signals by which the component communicates with the system. Yet it is more than a simple listing of such signals, for such signals are often preprocessed or "cleaned up" by the system. Thus, guarantees or

restrictions on such signals are also considered part of the environment. From the point of view of a component, *environment* and *system* are synonymous.

An analogy with software may be enlightening. A system tasked to perform calculations using the days of the year as input might contain twelve component modules which service each month. Erroneous dates such as October 35, April 0, and February 30 must be rejected. The system designer may decide that dates less than 1 or greater than 31 shall be rejected at the top level, but month-specific problems such as February 30 shall be handled by each specific month. Thus the environment guarantees to each month module that only dates in the range 1 to 31 will be passed on. The February module designer will incorporate checks to reject February 30 and 31. He will not incorporate code for negative dates because the system guarantees the module will not see them. In fact, the prudent module designer will exploit this fact to optimize his code. It is quite acceptable for the February module to produce all kinds of outlandish or "unspecified" behavior in the region that the system guarantees it will never reach.

Behavior. *Behavior* denotes what a component does or is required to do. When speaking in terms of behavior, one should avoid reference to any particular implementation of that component. Behavior has two aspects: *function* and *timing*.

Function. *Function* denotes the action of a component that transforms inputs into outputs. The speed at which this occurs is not considered part of the function.

Timing. *Timing* refers to the speed of a component, together with any ordering of events that may have to be enforced.

Supplier. The *supplier* is the institution that is legally responsible for the performance of a component. Normally, the supplier is the physical manufacturer. The component designer is usually an employee of the supplier, but not always.

Customer. The *customers* are those who procure and apply ICs that go into systems. This includes not only those who design and manufacture the original system, but also those who perform maintenance on such systems. Logistics agencies that procure spare and replacement parts are also customers.

1.1.3 Specification Models. In 1992, MIL-STD-454L, Requirement 64 (DoD, 1992), was amended to require that VHDL behavioral and structural models be delivered to the DoD for all newly procured military integrated circuits. In that same year, the Defense Electronics Supply Center (DESC) received funding from the Air Force's Producibility, Reliability, Availability and Maintainability (PRAM) office to develop a repository for these VHDL models (Noh, 1994). Such a role is the natural extension of DESC's traditional role of

(1) producing hard-copy specifications of military ICs, and

(2) maintaining design documents of devices supplied as "compliant" to these specifications.

The Defense Supply Center Columbus (DSCC) assumed the DESC mission after the two centers merged in 1996. In the language-based life cycle environment DSCC can expect to receive two kinds of VHDL models:

(1) Behavioral models and test benches to serve as specifications.

(2) Structural models to document compliant implementations.

Upon receipt of the models it is necessary for DSCC to approve or disapprove them on behalf of the government in some way, as DESC has done in the past with hard-copy documents. Of course DSCC is not the only organization to face such challenges.

Not everyone relies on military specifications. In fact, the DoD itself has ceased relying on formal military specifications (Perry, 1994). Those who do not rely on military specifications still need some vehicle to serve as a contract between customer and supplier. With the complexity of modern integrated circuits, the means of determining compliance to that contract must be formalized and automated as much as possible.

This discussion assumes for simplicity that the system designer issues a specification for each system component. Suppliers then seek to implement each specification. This does not mean that the system designer actually writes each specification. For "off-the-shelf" parts an adequate specification may already exist. For new parts, or for poorly specified parts, the designer may develop the specification himself. Nor will suppliers always blindly accept a specification. One can also think of a specification as a contract between a component and its environment (Stevens, 1994:126). Like any contract, the specification is often an object of negotiation.

VHDL behavioral models can be used for procurement, replacing hard-copy specifications. Thus, behavioral specification models will be used to verify the structural models that document implementations. One major goal of language-based design is to develop a formal specification that is faithful (in behavior and properties) to the original informal idea of the system designer. A second goal is to assure that the specification is "loose," expressing only the *required* properties of a device without unduly constraining the component design and manufacturing processes.

Many properties can appear in a specification model. These may include fan-in, fan-out, power consumption, physical size, pin arrangement, etc. Foremost in the mind of most designers and users, however, is the digital *behavior* (both *function* and *timing*)

of the device. The behavioral aspect is so dominant that most devices, such as "16-bit Adder" or "500 MHz Microprocessor," are overtly named accordingly to their behavior.

A VHDL model used for a specification is called a *behavioral model* because it uses the "behavioral" and not the "structural" constructs of the language. However, it must be understood that a specification model that expresses only the overt surface behavior of a device will be incomplete. Certain *invariant* properties—those expected to remain constant as execution proceeds—may be just as critical. As humans we tend to focus on the transformations or *changes* wrought by a device. Invariant properties may not be the focus of the initial specification and design effort, and thus may be overlooked. Webster's Ninth Collegiate Dictionary defines *invariant* as "unaffected by the group of mathematical operations under consideration." Examples of such invariant properties are *safety*, *liveness*, fairness, and *deadlock*.

Safety. *Safety* denotes the property that a device will not be presented with any inputs it is incapable of handling. As a constraint on the environment, *safety* is more often an obligation of the system than of the device.

Liveness. Whereas the "*safety* property claims that 'something bad' does not happen," the *liveness* property assures "that 'something good' eventually happens." (Manna and Pnueli, 1992:302). "Liveness properties deal with eventualities—events which must occur at some finite but unbounded time." (McMillan, 1993:5) In other words, *liveness* assures that the overt behavior we desire can be relied upon to complete within a finite time.

Fairness. The *fairness* property ensures "that a process, once initiated, will—sooner or later—get the opportunity to complete its actions" (Dijkstra, 1968). Another way to state this is that when a device needs some resource, it will eventually get

that resource. Thus, in a *fair* system, every system component or procedure will get a chance to execute. None will be "starved."

Deadlock. One also desires freedom from *deadlock.* A *deadlocked* system is one that reaches a state from which it can do no further transactions. In the common idiom, the system has "died" or "crashed." This can result, for example, from an "after you" situation, where two communicating components are each awaiting some action or acknowledgment from the other.

Function and *timing* are habitually included in a specification, but invariant properties may be overlooked. This is not surprising. Consider the property of *deadlock.* When hard-copy specifications were used, and the designer had to manually interpret those specifications, it was intuitively obvious that the customer did *not* want deadlocking parts. However, modern practice uses an electronic model for a specification, and an automatic synthesizer does the "interpretation." What if the specification model itself inadvertently deadlocks? The synthesis tool may faithfully implement that deadlock. The customer will have no legal recourse because deadlock was a property of the specification he supplied. Verification tools will not protect him, either, for they will simply report that the design and specification are "equivalent" or that the design "complies" with the specification.

Formal verification can be looked upon as the process of proving that invariant properties always hold. In fact, the desired behavioral transformation of a device, even though it expresses change, can be modified into an invariant form. This modified invariant form expresses the idea: "it is always true that we will get what we expect."

1.1.4 Implementation Models. A customer may or may not require design and construction *documentation* from the device supplier. Such documentation describes the design and construction of the supplied device in enough detail such that an alternate

supplier can readily remanufacture it. Commercial customers normally do not need such detailed documentation, and cannot get it anyway since many design and manufacturing techniques are considered proprietary. However the DoD, which must keep critical military systems running during dire emergencies, often requires the delivery of design and construction documentation for the purpose of remanufacturing in case the original supplier goes out of business.

Due to the complexity of modern microcircuits, customers will require design and construction documentation in the form of a *structural* model—a model that describes the device as a hierarchy of building blocks or modules, with only the leaf-level modules described behaviorally. Such a model is often an output of automatic synthesis techniques, where a device is built from a library of available subcomponents. Although a structural model has behavior, its behavior is not explicitly stated, but is the product of the combined behavior of its interacting subcomponents.

1.1.5 Compliance to Specification. With machine-readable models for both specification and implementation, one can use automatic methods to assure that an implementation is *compliant* to its specification. What does it mean for a component to *comply* to a specification? One way to achieve compliance is to require equivalence. The notion of hardware equivalence is not a trivial one. Examples of hardware equivalences abound (Bloom and Meyer, 1988; Brookes and others, 1984; De Nicola and Hennessy, 1984; Groote and Vaandrager, 1988; Hennessy and Milner.1985; Hoare, 1980; Milner, 1983 and 1989; Olderog and Hoare, 1986; Park, 1981; Phillips, 1987; Rounds and Brookes, 1981). Van Glabbeek has identified at least 14 distinct equivalence formalisms (van Glabbeek, 1990; 1990a). Total equivalence between component and specification is not desirable anyway because it is too restrictive. A good specification lays down no more restrictions than the system absolutely requires. It forms a behavioral

envelope within which the design must perform. The behavior of a good design will be "guard-banded" within that envelope. A compliant design will be capable of all the behaviors required by the specification, and none of the behaviors forbidden by the specification. However, additional behaviors, and even additional outputs, are permissible. In fact, deviations are even desirable, since the exploitation of "don't-care" states can improve the cost and efficiency of the design.

In the most general case, the implementation does not imply the specification, nor vice versa.   In the space of possible behaviors, neither set of behaviors will contain the other. To illustrate, consider the Venn diagram of Figure 1-1.
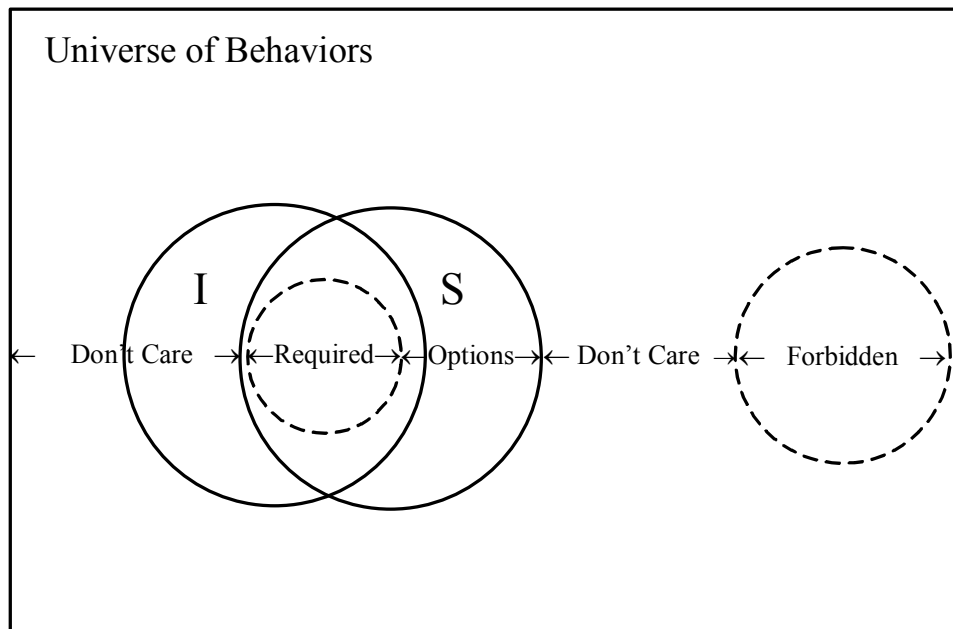
Figure 1-1. Compliance of *I* to *S*

The intersection of the specification *S* and the implementation *I* contains within it all the behaviors of *S* that are *required*. The area of *S* outside of the required behavior indicates behavioral options that the implementation can select from. An example of

such an option is an *output interleaving*—when the specification allows two or more signals to be generated in any order, but the implementation uses just one ordering. Outside of *S* and *I* will lie behaviors that are *forbidden*. Between *S* and the forbidden behaviors lie "don't-care" behaviors. The area of *I* outside of *S* represents additional behaviors that the implementation takes on in order to achieve efficiency.

1.2 Problem Statement

Equivalence is a mathematical notion that avails itself of such tools as deductive logic and rigorous proof. Thus, the tools of mathematics can be marshaled to determine the equivalence of digital circuits. With the advent of modern computers, these checks can be partially or totally automated—thereby eliminating human error and speeding up the process of microcircuit verification. However, the notion of compliance to specification is *not* an equivalence, as Figure 1-1 shows. The modern electronic design life cycle will benefit if the notion of hardware compliance can be placed on the same formal footing as equivalences. This will allow formal tools to evaluate the compliance of devices to specification models, and provide a means to validate the many transformations used during the design process. Such transformations ought to preserve a formal compliance property even when losing other properties. Therefore, the goals of this research are five-fold:

1. Determine the characteristics of a compliant device with respect to its specification. Study the expected behavior of an implementation in response to specified input, output and hidden action. Conversely, note any reverse obligations of the specification to implemented input, output and hidden action.

2. Incorporate this intuition into the formally-defined property of *congruent weak conformance* as a binary relation over processes. Make this formal property as "loose" as possible such that it admits all appropriate implementations and allows the most design flexibility.

3. Derive formal results for congruent weak conformance and related properties. Prove that congruent weak conformance is a partial order. Prove also that congruent weak conformance is fully substitutable in all contexts, is a valid model of safe substitution, and is indeed a *congruence*.

4. Outline the transformations necessary to create a semantic link from VHDL to CCS.

5. Show that such transformations are valid by proof that they preserve congruent weak conformance, thus allowing the more powerful verifications of CCS to accrue to VHDL models.

1.3 Organization of the Dissertation

Chapter 2 presents the prior art, describing process algebras (in particular *CCS*), modal and temporal logics, various equivalences, fixed points, and the techniques of *coinduction* and *transition induction*. Other process orders, preorders and partial orders from the literature are presented as potential competitors to congruent weak conformance. These ordering relations are discussed and their shortcomings noted.

Chapter 3 investigates the example of a binary-coded decimal (BCD) converter as a vehicle for extracting intuitive notions of device compliant. These notions are then

formalized by four transition rules, yielding a family of process relations called *weak conformations*. Weak conformations are precursor relations that will be refined in subsequent chapters. Formal results governing weak conformations will be presented as a series of propositions and proofs.

Chapter 4 presents *weak conformance* $\succeq_w$ (single underline) as the largest of the weak conformations. Further formal results governing $\succeq_w$ are proven in an effort to show it as fully substitutable, that is, a *congruence*. That attempt stalls pending additional refinement of $\succeq_w$ to a stronger property. Five constructional restrictions are then presented as requirements governing the building of specification and implementation models. These restrictions are shown to be reasonable constraints that do nothing more than codify good design intent. Congruent weak conformance $\succeqq_w$ is then defined as the $\succeq_w$ relation as refined by these restrictions. Additional results are proven, culminating in the demonstration that $\succeqq_w$ is both a partial order and a congruence, meriting the term *precongruence*. This establishes $\succeq_w$ as a correct model of safe substitution.

Chapter 5 presents a VHDL to CCS translation scenario. It starts with simple circuits of sufficient complexity to exhibit the semantics of VHDL while displaying the salient features of $\succeqq_w$. By comparing corresponding VHDL and CCS models, transformation rules are derived. Loss of information such as the explicit timing data of VHDL is noted. These transformations are then validated by proof that they preserve $\succeqq_w$. Thus safe substitution is preserved by these transformations despite the loss of other information.

Chapter 6 then presents conclusions and recommendations for future work.

For completeness, Appendix A gives a definition of *strong conformation* to contrast the *weak conformation* concept of Chapter 3. Strong conformation lacks the utility of weak conformation, and is not developed further.

Appendix B contains the longer proofs that would otherwise interrupt the flow of the dissertation, and Appendix C gives the transition laws for the process algebra CCS. Appendix D gives the "initial" VHDL models for the BCD-converter agents *S*, *I* and *J* given in Chapter 3. Appendix E gives the translated, or "target" VHDL model for the BCD-converter agent *S*.

II. Prior Art

This chapter discusses previous research and knowledge leading up to the present research. Section 2.1 presents the technique of *simulation*, both exhaustive *logic simulation* and its cousin *symbolic simulation*. *Formal verification* and its two main traditions: theorem proving and model checking, is the subject of Section 2.2. *Process algebras,* the languages used to model concurrent hardware, follow in Section 2.3. This dissertation employs the process algebra known as the *Calculus of Communicating Systems* (CCS) (Milner, 1989). CCS is used to introduce the idea of *equivalence* of two process algebraic models (*i.e.* circuits) in Section 2.4. Of the many hardware equivalences, four appear here. The last, *observational congruence,* is the appropriate equivalence that captures the notion of *safe substitution*.

Section 2.5 introduces the modal and temporal logics, which have the power to identify equivalences and other properties of process agents. These logics are then extended, by means of the fixed-point notation, into the modal μ calculus. The *Concurrency Workbench* (Cleaveland and others, 1989; Cleaveland and Parrow, 1993) is introduced as a tool whose notation greatly simplifies the ungainly notation of the modal μ calculus, and allows one to investigate properties of CCS agents using the power of *bisimulation* (Park, 1981), a semantics capable of stricter verifications than the simulation semantics of VHDL.

Proofs of process algebraic propositions often require the technique of *transition induction* (Milner, 1989:58, 100). Transition induction is a variation of *coinduction* (A. Gordon, 1995; Rutten, 1996; Jacobs and Rutten, 1997; Wegner and Goldin, 1999). Coinduction is not as well known as mathematical induction. Therefore the general techniques of coinduction and transition induction are introduced in Section 2.6.

Sections 2.7 and 2.8 review the work of researches whose aims appear similar to those of the present research. These contributions fall into two camps. Section 2.7 reviews those efforts that link languages such as VHDL into formal methods. Section 2.8 reviews various process ordering, preordering and partial ordering relationships.

2.1 Simulation

The *VHDL Language Reference Manual* refers to an event-based simulation cycle to define its constructs (IEEE, 1993). Thus, a set of informal simulation semantics is assumed for VHDL (van Tassel, 1994). This is consistent with the current practice that verifies designs by computationally simulating the operation of the design. *Simulation* involves the submission of test stimuli to the device model while observing the responses. Ideally, the designer will simulate both the design model and the specification model, checking that they yield the same results under simulation.

A VHDL language-based design environment will represent the original requirements (specification) using two models: (1) a VHDL behavioral device model, and (2) a VHDL *test bench.* The first represents the specified device as a finite set of behaviors. It describes the transformation of inputs to outputs, the ordering of events, and speed requirements. Ideally, this model should use only the behavioral constructs of the language. It should not suggest an internal structure that may unduly limit possible implementations. The *test bench* is a VHDL model used to exercise the device model during simulation. It contains the behavioral device model as a component. The test bench includes a set of *test vectors* that represent input stimuli and the expected output responses. During simulation, the test bench submits the input portion of each test vector to the device model and compares the resulting output with the expected output. The behavioral device model must be able to pass this simulation before a specification can be released.

Later, when the design of a potential implementation arrives, a structural VHDL model that represents the submitted design replaces the behavioral device model. Ideally, this model is synthesized from the design by automatic means. The simulation is then repeated on the implementation model. If this model also passes the simulation, then the design has been *validated* to be a correct implementation of the specification.

For simulation to be totally effective, it needs to embody all possible behaviors. In other words, the test vector set must be exhaustive both in all legal timing variations as well as behavioral variations. Unfortunately, the complexity of modern integrated circuits militates against an exhaustive test vector set. The number of possible test vectors is exponential on the number of inputs. Exhaustive simulation is intractable for modern designs. Actual simulations rely on a limited set of test vectors and thus do not give complete verification assurance.

Symbolic simulation is related to logic simulation (McMillan, 1993:126). In ordinary logic simulation the test vectors consist only of the binary constants **0** and **1**. In symbolic simulation a vector can consist of Boolean variables and functions as well as constants. Thus, multiple specific instances of behavior can be abstracted away and verified as a class.

2.2 Formal Verification

As an alternative to simulation, researchers have investigated the use of formal methods to verify hardware. *Formal verification* seeks to establish the correctness of designs by means of mathematical proof (McMillan, 1993:1).

There are two common formal verification traditions, *theorem proving* and *model checking* (McMillan, 1993:2). The theorem proving approach models the device and its specification in a formal logic. The device model constitutes the "axioms" of a formal system. This approach then seeks to construct a proof leading from the axioms to the

specification. In other words, the device model should logically *imply* the specification. Unfortunately, these proofs can be lengthy, and the process is not fully automated.

The model checking approach is also a proof based approach, but it is restricted so that full automation *can* be achieved. During model-checking, the device is modeled specifically as a finite state machine, and specifications are written as logical assertions to be proved about that specific finite state machine.

Formal verification by model checking and VHDL validation through simulation share a similarity. Both approaches use a two-part specification. The first part is a model of the proposed device as a finite set of behaviors. Requirements for the device model to meet appear separately. For a VHDL simulation-based environment, requirements are embodied in the test bench as a voluminous set of test vectors.   In the model-checking environment, requirements are expressed more concisely as a set of logical assertions.

2.3 Process Algebras

The subtle properties of *liveness, fairness,* and *deadlock* become issues when dealing with the parallelism and concurrency inherent in structural models. Consider that a truly non-parallel uniprocessor would have no means of deadlocking unless it were designed with an explicit *HALT* instruction. Unfortunately, such a processor is only an abstraction. Digital electronic hardware, being made up of physical components that operate in real time, is inherently concurrent and parallel, and thus deadlock is a possibility.

Process algebras such as the *Calculus of Communicating Systems* (CCS) (Milner, 1989) and *Communicating Sequential Processes* (CSP) (Dijkstra, 1968; Hoare, 1985) are used to model concurrency and thus are useful for modeling digital electronic

hardware.[1]   CCS, for example, is a very concise, but highly expressive language. It has mechanisms for both behavioral and structural modeling. Especially important are its mechanisms for expressing non-deterministic choice and hidden internal action. Processing elements are known as *agents*, and are often recursive.

Consider the asynchronous communication device called the *C* element (Shams and others, 1998). The *C* element awaits the arrival of two *concurrent* inputs *a* and *b*. Once both have been received, an output $\overline{c}$ is produced. Equation 2-1 is the CCS model of the *C* element's behavior:

$$C \stackrel{def}{=} ab\overline{c}.C \; + \; ba\overline{c}.C \tag{2-1}$$

Being concurrent, the input signals *a* and *b* may arrive in either order. This allowance is indicated by alternative execution branches separated by '+'. No matter which branch is selected, the output $\overline{c}$ is emitted after inputs *a* and *b* are received. Following the output, the agent returns to the initial state *C*, ready to receive more inputs.

CCS *processes* (or *agents*) appear in upper case. Lower case names serve as transition labels, with output transitions bearing an overbar.[2]   There are six CCS combinators or operators:

- The *Constant* operator, '$\stackrel{def}{=}$', which assigns an agent name to a behavior.

- *Prefix*, denoted by the period, to indicate one action following another.

- *Choice* or *Summation*, denoted by '+', to indicate a fork in the execution path.

---

[1] Process algebras are also commonly called *process logics*, but this dissertation maintains a distinction. "Process algebra" is reserved for languages that represent closed systems of processes and operations that transform them. "Process logics" are systems that manipulate predicates defined over process algebras.
[2] When an overbar is typographically difficult it is customary to use a leading "tic" mark: $'c$.

- *Parallel Composition*, denoted by '(*A* | *B*)' to indicate agents *A* and *B* operating concurrently.

- *Relabeling.* The notation '*E*[*x*/*y*]' indicates that transition *x* has been renamed to *y* in the agent *E*.

- *Restriction*, denoted by the backslash character '\'.

To accomplish *behavioral* modeling in CCS, the first three combinators: Constant, Prefix and Choice,[3] suffice. Furthermore, the Choice operator can also be used to express non-deterministic behavior. Consider the agent:

$$COIN \stackrel{def}{=} flip.\overline{heads}.COIN + flip.\overline{tails}.COIN \qquad (2\text{-}2)$$

Here the environment can exert no control over the outcome, since the input *flip* occurs in both branches. Once a *flip* arrives, the *COIN* agent non-deterministically selects one branch, and produces an output accordingly. In this instance + involves an *internal*, or *non-deterministic* Choice. However, + is not always the harbinger of non-determinism. The *external* Choice expressed in the *C* element specification (Equation 2-1) is perfectly predictable due to the environment's ability to control the input sequence and select which branch is executed.

Like the *C* element and the *COIN* agent, most useful CCS agents are recursive in their behavioral description. Real hardware agents do not simply compute some result and terminate. They more closely resemble what Manna and Pnueli call *reactive programs* (Manna and Pnueli, 1992:vii). Rather than halting, they forever await inputs from their environment, respond, and then wait again.

---

[3] Consistent with Milner's usage, combinator names are capitalized to distinguish them from their common English meanings.

The last three combinators: Parallel Composition, Relabeling and Restriction, add the ability to perform *structural* modeling. As an example, consider the behavior of a simple one place buffer or *FIFO*:

$$FIFO \overset{def}{=} in.\overline{out}.FIFO \qquad (2\text{-}3)$$

One can build a two-place FIFO by connecting two one-place FIFOs in series, as shown in Figure 2-1.

Figure 2-1. Two-place FIFO

This composite construction *FIFO2* is modeled in CCS as follows:

$$FIFO2 \overset{def}{=} (\ FIFO[mid/out] \mid FIFO[mid/in]\ )\backslash\{mid\} \qquad (2\text{-}4)$$

The Parallel Composition (FIFO…| FIFO…) denotes the building of a composite model from two submodels—in this case two identical FIFOs. The Relabeling functions [*mid/out*] and [*mid/in*] then rename two of the ports to $\overline{mid}$ and *mid*. This forms an implicit internal connection. Any communication between the components of a Parallel Composition is accomplished when *actions* and *co-actions* share the same label, differing only by the overbar. In the parlance of hardware description languages, Relabeling expresses the "named association" of an *actual* signal *mid* to *internal* signals *in* and *out*.

The Restriction mechanism \{*mid*} in turn hides the internal signal *mid* from the external environment. The signal ceases to be a port to the outside world. In that sense the Restriction operator models abstraction by hiding a lower-level detail. This hiding of internal signals, which is implicit in hardware description languages such as VHDL, must be stated explicitly in CCS by means of the Restriction mechanism.

In its treatment of internal action, CCS differs significantly from languages such as VHDL where internal action is not represented at higher levels of abstraction. In CCS, such hidden action is denoted by the symbol τ. *All* silent actions are abstracted into this single symbol. Practitioners will sometimes use a subscript such as τ_{mid} to track the origin of these actions, but the subscript is semantically meaningless within the context of CCS.

After expanding each *FIFO* and Relabeling their ports, one can rewrite *FIFO2* as

$$FIFO2 \overset{def}{=} (in.\overline{mid}.FIFO | mid.\overline{out}.FIFO)\backslash\{mid\} \qquad (2\text{-}5)$$

where $\equiv$ is syntactic identity. Using a single-shaft labeled arrow to denote atomic transitions one writes

$$FIFO2 \overset{in}{\longrightarrow} (\overline{mid}.FIFO | mid.\overline{out}.FIFO)\backslash\{mid\} \qquad (2\text{-}6)$$

At this point the renamed signals *mid* and $\overline{mid}$ can communicate or synchronize, and the next transition is a τ.

$$(\overline{mid}.FIFO | mid.\overline{out}.FIFO)\backslash\{mid\} \overset{\tau}{\longrightarrow} (FIFO|\overline{out}.FIFO)\backslash\{mid\} \qquad (2\text{-}7)$$

The compound agent conducts a silent action by transferring a datum from the first to the second *FIFO*. Though unseen, this hidden action does indeed affect the behavior of the compound agent. The evolving *FIFO2* cannot accept a second *in* action until this internal transfer occurs.

Because the composite agent *FIFO2* has a depth of two, the user will expect it to accept two inputs before an output is issued. Or, if an output is issued after one input, one will expect that *FIFO2* is now empty and can accept two more inputs. Therefore, one behavior the user *expect*s is $in.in.\overline{out}$. He does not expect to be delayed at all if he wishes to send two symbols *in* in succession. Yet since the agent *FIFO2* cannot accept a second *in* until the internal action has transpired, the behavior he actually *gets* is $in.\tau.in.\overline{out}$.

Some might argue that the τ interruption is inconsequential. In real hardware such internal actions occur readily enough and for practical purposes they can be ignored. Others might argue that unless one knows the target technology and how the circuit will be laid out, it is wiser to assume no more than necessary about any delay associated with internal actions. Designers of *synchronous* circuits, in particular, eliminate the need to consider internal action by calculating worst-case delays and then slowing down the system clock to insure no internal actions are pending when the clock advances. *Asynchronous* designers however, who use no clock, must take note of internal action in some way. For them, the τ mechanism in CCS is very powerful.

Differences in opinion about how to handle internal actions (and the conditions under which they must be respected or ignored) give rise to various *hardware equivalences*—covered in the next section.

2.4 Hardware Equivalences

An equivalence relation divides a set into *equivalence* classes. *Within* each

equivalence class all members are *equivalent* and *between* equivalence classes all members are distinct. The *weakest* possible equivalence relation simply declares all members of a set $\mathcal{P}$ to be equivalent. The *strongest* possible equivalence, on the other hand, distinguishes every member of $\mathcal{P}$, placing each into its own (singleton) equivalence class. Equivalence relations in process algebras can also be characterized by their *strength* with the stronger making finer distinctions among agents and the weaker identifying more agents. Modeling accuracy favors stronger equivalences, whereas design flexibility favors weaker equivalences.

Van Glabbeek has listed eight semantic criteria that characterize various hardware equivalences [vG90a]. The four main criteria are:

- *Linear* time versus *branching* time. Linear time semantics distinguishes processes based on the content of their observable runs, whereas branching time semantics maintains information where different courses of action diverge.

- *Interleaving* semantics versus *partial-ordering* semantics. This distinction relates to the expression of concurrency. In interleaving semantics there is only "liveness on a symbol." CCS is an example of interleaving semantics. Since symbols are only issued one at a time the concurrency between two symbols must be expressed by explicitly giving the interleavings, for example, *a.b + b.a.* In partial order semantics, also known as "true concurrency," there is "liveness on symbols and transitions." Van Glabbeek lists the Petri Net discipline as an example of true concurrency. When a transition is live it can fire and release multiple tokens without specifying the interleavings among those tokens.

- *Abstraction* of internal action. When equating agents, internal actions can be totally ignored, or taken into account in various ways.

- Treatment of *infinite processes*.

Van Glabbeek has identified 14 equivalences (van Glabbeek, 1990; 1990a). Four such equivalences are presented here: *trace equivalence, strong equivalence, observational equivalence,* and *observational congruence.*

     2.4.1 Trace equivalence. Two agents, *P* and *Q*, are *trace equivalent* when every sequence of visible actions produced by *P* is producible by *Q*, and vice versa (Milner, 1989:204).

Definition 2-1. Process agents *P* and *Q* are *trace equivalent*, written $P \sim_t Q$, if $\forall s \in \mathcal{L}^*$, $P \overset{s}{\Rightarrow}$ iff $Q \overset{s}{\Rightarrow}$.

This is a common sense version of equivalence, but is often too weak for many purposes. Consider the agents:

$$Y \overset{def}{=} a.(b.NIL + c.NIL) \tag{2-8}$$

$$V \overset{def}{=} a.b.NIL + a.c.NIL \ ^4 \tag{2-9}$$

The two agents *Y* and *V are* indeed trace equivalent, sharing the trace set {*a, a.b, a.c*}. However their observable behaviors are not the same. The agent *Y*, after performing an *a* action, still has the option to perform either *b* or *c*. For the *V* agent however, this choice is taken away. Upon receipt of the *a*, the *V* agent non-deterministically chooses a branch, evolving to either *b.NIL* or *c.NIL*, after which it will reject either *c* or *b*, respectively.

     2.4.2 Strong equivalence. The notion of strong *bisimilarity* addresses this difference in observable behavior between trace equivalent agents (Milner, 1989:88). Two agents, *P* and *Q*, are said to be *strongly bisimilar* if each can perform all the actions

---

[4] *NIL* is a special CCS agent that can do no actions, and can be considered a *HALT.*

of the other and, after every such action α, the immediate successor agents (α-derivatives), P' and Q' are themselves strongly bisimilar. Strong bisimulation is thus a binary relation among agents. Formally, a strong bisimulation, S, satisfies the so-called "back-and-forth" property:

Definition 2-2. A binary relation S among processes is a *strong bisimulation* if $\forall$action α

(i) Whenever $P \xrightarrow{\alpha} P'$ then $\exists Q'$ such that $Q \xrightarrow{\alpha} Q'$ and $P' \, S \, Q'$.

(ii) Whenever $Q \xrightarrow{\alpha} Q'$ then $\exists P'$ such that $P \xrightarrow{\alpha} P'$ and $P' \, S \, Q'$.

Many relations satisfy Definition 2-2, including the empty relation. However the empty relation is not useful, since it equates no agents. One normally wishes to equate as many agents as practical. Therefore, one prefers the *largest* strong bisimulation $\sim$, which is called *strong equivalence*. Strongly equivalent agents each match the actions of the other, including the internal action τ. The two agents V and Y given above, though trace equivalent, are *not* strongly equivalent. Agent V has two a-derivatives, $b.NIL$ and $c.NIL$. Neither of these can perform *all* the actions of the single a-derivative of Y, $b.NIL + c.NIL$, which can perform both b and c.

2.4.3 Observational equivalence. The notion of strong equivalence is too strong for many purposes. For example, strong equivalence distinguishes between the agents $a.NIL$ and $a.\tau.NIL$:

$$a.NIL \,\nsim\, a.\tau.NIL \qquad\qquad (2\text{-}10)$$

However such a distinction normally makes little difference to users. After receiving the a action, both agents eventually evolve to NIL and halt anyway.

Therefore, the notion of strong bisimilarity is weakened to *weak bisimilarity* (often called simply *bisimilarity*), which abstracts away τ actions (Milner, 1989:108). Weak bisimulation also obeys a "back-and-forth" property in a manner analogous to strong bisimulation:

Definition 2-3. A binary relation $\mathcal{B}$ among processes is a *weak bisimulation*, if $\forall$ action α

(i) Whenever $P \xrightarrow{\alpha} P'$ then $\exists Q'$ such that $Q \xRightarrow{\hat{\alpha}} Q'$ and $P' \mathcal{B} Q'$.

(ii) Whenever $Q \xrightarrow{\alpha} Q'$ then $\exists P'$ such that $P \xRightarrow{\hat{\alpha}} P'$ and $P' \mathcal{B} Q'$.

The *hat* embellishment $\wedge$ above an action or an action sequence removes τ actions from that sequence. Since α above represents single, or *atomic* actions, the hat notation changes a τ action to the *empty sequence*, ε. All other actions are unchanged. The double-shafted arrow $\Rightarrow$, on the other hand, allows insertion of any number of τ actions necessary to complete a transition. Thus, weak bisimulation differs from strong bisimulation in that any one agent can match the τ actions of the other with zero or more τ actions. As was true with strong bisimulation, there are also many weak bisimulation relations. Again, the *largest* weak bisimulation $\approx$, called *observational equivalence*, is the most interesting. Note that $\alpha.\tau.NIL \approx \alpha.NIL$. These two agents are *not* distinguished under $\approx$, as they are under $\sim$.

2.4.4 Observational congruence. A motivation for finding equivalent hardware agents is to safely substitute one for the other. This safe substitution property is known as *congruence*. A *congruence* is a relation that is preserved by every operation of the underlying algebra. Alternately, one can say that a congruence is preserved by all *contexts*. Consider the Prefix operation. If $P \approx Q$ then one hopes that $a.P \approx a.Q$. In other words, the Prefix operator ought to preserve $\approx$. This is one congruence law. Under CCS there are six congruence laws, one for each CCS operator. Unfortunately, observational

equivalence $\approx$ fulfills only five of the six laws. It is not preserved by Summation. For example:

$$\tau.a.NIL \approx a.NIL \tag{2-11}$$

$$b.NIL \approx b.NIL \tag{2-12}$$

yet,

$$\tau.a.NIL + b.NIL \not\approx a.NIL + b.NIL \tag{2-13}$$

because the initial τ action of the left hand agent can preempt the choice allowed by '+'. The left-hand agent is therefore *unstable* (Milner, 1989:112). When the τ occurs, the left-hand agent's ability to perform *a* spontaneously evaporates, without the occurrence of a visible action. Meanwhile the right-hand agent can still perform either *a* or *b*. Clearly, there is a difference between the "safe" τ appearing in $a.\tau.NIL$ versus the preemptive τ appearing in Equation 2-13. That difference is *guardedness* (Milner, 1989:65).

Definition 2-4. *X* is *guarded* in an expression *E* if each occurrence of *X* within *E* lies within some subexpression $\ell.F$ of *E*, where ℓ is a visible action. $\ell \neq \tau$.

Only *visible* actions can serve as guards. *X* is therefore guarded whenever some visible Prefixed action must always be encountered before the execution can proceed to *X*. Thus, the τ in $a.\tau.Nil$ (which can safely be ignored) is seen to be *guarded*. The *unguarded* τ in Equation 2-13, however, creates an instability and destroys the congruence of $\approx$ under Summation. Hence, the property of observational equivalence is modified to that of *observational congruence* (Milner, 1989:153). Observational

congruence obeys a "back-and-forth" property similar to weak bisimulation, with the hats removed from the action symbol α.

Definition 2-5. Process agents P and Q are *observationally congruent,* written $P = Q$, if $\forall$action α:

   (i) Whenever $P \xrightarrow{\alpha} P'$ then $\exists Q'$ such that $Q \xRightarrow{\alpha} Q'$ and $P' \approx Q'$.

   (ii) Whenever $Q \xrightarrow{\alpha} Q'$ then $\exists P'$ such that $P \xRightarrow{\alpha} P'$ and $P' \approx Q'$.

   Observational congruence is also called *equality*, and denoted $P = Q$. Under observational congruence, initial, unguarded τ actions must be matched τ for τ. However, Definition 2-5 does *not* demand the α-derivatives P' and Q' in turn to be $=$, merely $\approx$. Otherwise guarded τ actions would eventually pop out and be evaluated as unguarded. Thus, observational congruence continues to abstract away *guarded* τ actions in the same manner as $\approx$, while respecting unguarded τ actions. Milner shows that $=$ is a slightly stronger equivalence than $\approx$, which can be derived by restricting $\approx$ to initially stable agents (Milner, 1989:Proposition 5-9).

2.5 Modal and temporal logic

   The *Edinburgh Concurrency Workbench*, or *CWB* (Cleaveland and others, 1989; Cleaveland and Parrow, 1993) accepts agents described in CCS and test them against various equivalences. Thus, one can present a specification as a behaviorally described CCS agent. A candidate implementation can be presented by assembling its components using the Parallel Composition operator. Specifications and implementations are compared for equivalence. However, such verifications can take a very long time if the agents have many states. In many instances, it is more convenient to check agents to see if they satisfy certain defined logical properties.

2.5.1 Hennessy-Milner Logic. The CWB can also check assertions written in the *Hennessy-Milner Logic* (HML) (Stirling, 1992). Since it deals with assertions about processes, HML can be called a *process logic*. Being a logic, it includes the standard Boolean connectors $\neg, \wedge, \vee, \Rightarrow$, T and F. The notation $P \models A$ means that CCS agent P *satisfies* the HML assertion A.

HML is also a *modal* logic, able to express assertions that are *possibly* true or *necessarily* true. It includes *modal quantifiers*. Angle brackets denote *possibility* and square brackets denote *necessity*. These brackets enclose actions or sets of actions. Thus the notation $P \models [a]A$ means that for *every a* action that agent P can perform, the successor agent $P' \models A$. Note that if P *cannot* perform such an *a* action then $P \models [a]A$ is vacuously true. $P \models <a>A$ means that there is *at least one a*-action that P can do such that the successor $P' \models A$. The special identifier '-' denotes the entire set of actions. Thus $P \models [-]A$ means that *every* action performable by P results in an agent satisfying A.

HML gives the ability to define and check properties of agents on the CWB. First note that *all* agents satisfy the trivial assertion T (truth). One can construct the assertion $E \models <->T$ which says that there is *some* action that agent E can perform and evolve to *something*. In other words, E can perform *some* action and is therefore *live*. Conversely, $E \models [-]F$ says that E cannot do *any* action and is therefore *deadlocked*. Thus the CWB provides a means of defining and checking properties such as deadlock and liveness.

2.5.2 Fixed Points. For the modeling of digital hardware one generally desires *reactive* agents that operate indefinitely, continually returning to a "ready" state. In other words, the most interesting agents are *recursive*. Thus, interesting propositions about such agents will also be recursive. To handle such recursive propositions, HML is augmented with the ability to handle *fixed points*. The result is the *modal μ calculus* (Stirling, 1992). Consider again the recursive C element:

$$C \overset{\text{def}}{=} a.b.\bar{c}.C + b.a.\bar{c}.C \qquad (2\text{-}14)$$

A property one might suspect for the C element is that any output action $\bar{c}$ is preceded by exactly two input actions. Thus:

$$C \models <\text{-}><\text{-}><\bar{c}>T$$

$$C \models <\text{-}><\text{-}><\bar{c}><\text{-}><\text{-}><\bar{c}>T$$

$$C \models <\text{-}><\text{-}><\bar{c}><\text{-}><\text{-}><\bar{c}><\text{-}><\text{-}><\bar{c}>T$$

$$C \models <\text{-}><\text{-}><\bar{c}><\text{-}><\text{-}><\bar{c}><\text{-}><\text{-}><\bar{c}><\text{-}><\text{-}><\bar{c}>T \qquad (2\text{-}15)$$

and so on. This sequence suggests a more compact, single assertion:

$$X \models <\text{-}><\text{-}><\bar{c}>X \qquad (2\text{-}16)$$

Such a formula is a recursion of the form $X = F(X)$. A set of system states X that can satisfy this recursive assertion is called a *fixed point* solution because it is unaffected by repeated applications of F. There may very well be more than one set of states that qualifies as a fixed point solution. These solution sets are partially ordered under the subset relation $\subseteq$. The smallest such set is called the *minimum* fixed point. It contains only those states for which the assertion is *necessarily* true. The largest such set, the *maximum* fixed point, includes all states *except* those for which the assertion is necessarily false. The minimum and maximum fixed points of the formula $X = F(X)$ are denoted *min(X.F(X))* and *max(X.F(X))*, respectively.

2.5.3 Temporal Logic. A class of logics called *temporal* logics "defines predicates over infinite sequences of states" of systems as they evolve over time (Manna and Pnueli, 1992:179). The modal μ calculus qualifies as a temporal logic. However, the complicated fixed point notation of the modal μ calculus can be very difficult to follow. A refinement to the calculus defines certain temporal operators which amount to a shorthand for more extensive modal μ fixed point formulas. These operators have simple English interpretations. The operator □ can be read as "always." It precedes an assertion that is guaranteed to hold at all future times, regardless of how the behavior may branch. $P \models \Box E$ means that E holds for all future successors to agent P. Read as "always E," $\Box E$ is easier to interpret than the more ungainly $max(X = E \wedge [-]X)$. The diamond operator, $\Diamond$, denotes possibility. $P \models \Diamond E$ means that there is *some* execution sequence for which at least one successor satisfies E. The operator *EV* denotes *eventuality.* $P \models$ EV E means that along *all* execution branches, sooner or later, a successor agent will be encountered which satisfies E.

These convenient operators can be defined as macros in the CWB. Conventional usage defines the macros: *BOX*, *POSS*, and *EV* to denote the operators: □, $\Diamond$, and *EV*, respectively. Liu has shown how this calculus can be used to check for specification properties on the CWB (Liu, 1992). The *C* element, for example, should satisfy the property that, after both inputs *a* and *b* have been received the only possible move is an output $\bar{c}$. Liu has built a macro called *ONLY'c* defined as meaning $<\bar{c}>$ & $[-\bar{c}]F$ ("You can always perform $\bar{c}$ but nothing else is possible"). Applied to the *C* element, Liu derives the specification

$$\Box (\ \Box [a][b]\ ONLY'c\ )\ \&\ (\ \Box [b][a]\ ONLY'c\ ) \qquad\qquad (2\text{-}17)$$

which literally says that it is always the case that every time *a* is followed by *b,* or *b* by *a,* the only possible action is an output $\bar{c}$.

2.6 Coinduction and Transition Induction

Since recursive CCS agents never halt, such agents produce action streams that are infinite in extent. These streams can be *observed* from one end by unwinding the agent definition. However, one could never proceed to *construct* such a stream from the empty sequence ε. The fixed point approach presented above allows one to reason about these infinite streams. Modal and temporal logics which incorporate fixed point reasoning can be automated by tools such as the Concurrency Workbench to reason about recursive agents.

For manual proofs, a technique called *coinduction* is employed to reason about infinite streams and recursive processes (A. Gordon, 1995; Rutten, 1996; Jacobs and Rutten, 1997; Wegner and Goldin, 1999). Coinduction is the dual of the more familiar *induction* technique. Both techniques can be used both to conduct proofs and to provide definitions. The differences between the two techniques can best be highlighted by examining how (co)inductive definitions are pursued.

An inductive definition consists of three general parts, with the third so obvious that it is usually not stated. The three parts of an induction are: (1) the *basis*, (2) the set of *constructors*, and (3) the principle of *minimality*. The *basis* is a starting point from which to build the set being defined. For the natural numbers **N** the basis is the number 0. *Constructors* are means to build other members of the defined set. A single constructor + (or alternately the successor function $S(x) = x + 1$) suffices for **N**. The principle of *minimality* asserts that nothing else fills the definition except what can be constructed from the basis via the constructors. One can informally state the inductive definition of **N** as

Basis. "0 is a natural number."

Construction. "Successors of natural numbers are natural numbers."

Minimality. "Nothing else is a natural number."

Coinduction consists of two parts instead of three, since it lacks an analog for the *basis*. In place of *constructors*, coinduction uses *observers*—means of observing the *behavior* of the item being defined. One is generally unaware of the *structure* of the entity that produces said behavior; only the behavior itself is accessible. In place of *minimality*, coinduction uses a principle of *maximality*. Whereas minimality *forbids* everything that cannot be constructed, maximality *allows* everything that is not forbidden by the observations.

Consider, once again, the *C* element. The attempt to define a *C* element inductively will fail. One might propose the *NIL* agent as a basis and use the CCS operators as constructors; but one cannot build a recursive agent from NIL. Similarly, one cannot build the associated infinite action streams a, b and $\overline{c}$ by construction from ε. Rather, one has only the recursive behavior of C: $C \stackrel{def}{=} a.b.\overline{c}.C + b.a.\overline{c}.C$. An appropriate *observer* then is a rule that accesses the head of the behavior and defers the evaluation of the rest. An observer function may look something like this: $behavior(\alpha.X) = \alpha.behavior(X)$. Observers can be applied arbitrarily many times to unwind more and more behavior, but the end of the behavioral streams can never be reached. The coinductive definition of the *C* element can be informally stated as

Observation. "The *C* element can perform all action streams that unwind from
$$C \stackrel{def}{=} a.b.\overline{c}.C + b.a.\overline{c}.C."$$

Maximality. "Everything consistent with this observation is a *C* element."

Coinductive *proof* can appear circular and unsatisfying due to its lack of a *basis* step. For a coinductive proof it suffices to shown that a single unwinding of each observer function preserves the property in question. For CCS agents, the CCS transition rules (Milner 1989:45, 57) serve as observers, and proofs of properties over CCS agents are often coinductions employing these observers. Milner calls such coinductive proof *transition induction* (Milner, 1989:58, 100) and reaches conclusions "by induction." *Transition coinduction* might be a more appropriate term, but Milner's usage predates the general recognition of coinduction as a technique distinct from induction.

2.7 Applying Formal Methods to VHDL

This section presents the efforts of researchers who have studied, more specifically, the formal verification of VHDL models. These efforts fall into two general camps. First are the *extraction* techniques that seek to recover higher-order function from low level or "flat" models. Extraction includes both *logic extraction,* in which high-order structural blocks are substituted in flat structural models, and *temporal extraction*, where a more general model of behavior is substituted for a collection of simpler behaviors. The second camp seeks to translate between VHDL and other languages or tools so that the power of those tools will accrue to VHDL models. In the second camp, an understanding and defining of VHDL semantics is essential.

2.7.1 Extraction. The extraction process is one of iterative substitution using templates. The extractor repeatedly examines a flat design for subunits that match some template. Whenever such a match is found, those subunits are deleted and replaced with a single, equivalent, higher-level unit as dictated by the template. For example, one knows (or at least believes) that three NAND and two XOR gates, when connected as shown in Figure 2-2, will create a one-bit full adder. Similarly, eight full adders in cascade will form a byte-wide adder. By such repeated substitution, one may find that a
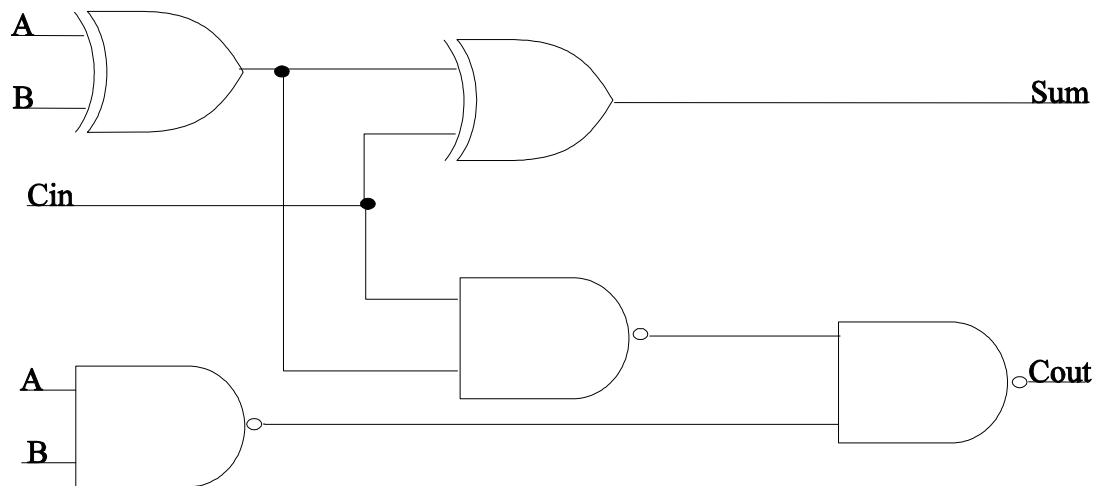
Figure 2-2. Full Adder

network of interconnected gates can be transformed, say, to a 32-bit ALU. Extraction thus serves as a verification that the original flat design is "equivalent" to the 32-bit ALU. Extraction presumes that the equivalence notion used to govern the substitutions is in fact a congruence.

2.7.1.1 Logic Extraction with VHDL. Dukes applied the process of logic extraction to VHDL models in the development of his *Generalized Extraction System* (GES) (Dukes, 1993). The extraction process itself is only as accurate as the templates used. Dukes realized that in a controlled VHDL-based design environment where the design library itself was developed and documented with VHDL, there was no need to produce these templates manually. Rather, extraction templates, or *extraction rules* could be automatically derived from VHDL structural models of library components. His GES system, written in Prolog, would first derive appropriate extraction rules from VHDL models or the technology design library, and then apply those rules to perform extractions on circuits designed in that technology.

One limitation of Dukes' technique, and indeed of logic extraction in general, is the strict dependence on structural templates. It does not seek to establish the behavioral equivalence of models. Even when, as with GES, the extraction rules are derived from a

design library, the extraction system takes for granted the library designer's claim that, for example, "these five gates are equivalent to one full adder." It assumes a behavioral equivalence between the gates and their purported function. Hopefully, library units *will* have been independently and exhaustively verified. For a component as simple as a five gate device, this is probably true. However, one's confidence becomes less certain as when moving up in complexity.

Dukes' tool reacts only to structural VHDL constructs. Any behavioral constructs, such as a process statement or an *assert*, is ignored. Thus, GES and logic extraction in general, will not verify a design against some purely behavioral device model, nor will it verify that logical conditions are met.

2.7.1.2 Temporal Extraction. Fujita developed a Prolog-based temporal extractor (Fujita and others, 1983; 1983a). This tool extracts and verifies temporal formulas using the "temporal logic decision procedure" developed by Wolper (Wolper, 1981). Thus, Fujita's tool is a *behavioral* extractor, in contrast to Dukes' *structural* extraction system. Fujita aimed to verify that a collection of behaviors of some circuit would satisfy some desired "protocol" (in other words: "higher-level behavior").

Fujita notes that, in general, the satisfiability of temporal formulas is undecidable. However, he draws on Wolper's method, which uses rewrite rules based on a right-linear grammar. Wolper had 14 such rules, which tend to transform other temporal operators into *next* operators, 'O', and then migrate these operators to the left of the formula. At any instant in time Wolper needed only to deal with 'O' since any other temporal operator would be embedded within the formula, to eventually pop out as 'O' anyway. Thus, instead of dealing with the undecidability associated with infinite sequences of states, Wolper's method looked forever at only the "next" state or event. Note how this quite naturally mimics the VHDL simulation engine, wherein the simulator focuses on

the next scheduled event, ignoring any other transactions until they in turn become the "next event."

Fujita's reliance on the Wolper procedure means that his method, when applied to VHDL, only captures its simulation semantics. Furthermore, just as logic extraction compares only structural models, Fujita's temporal extraction technique compares only behavioral models. Needed is a method of verifying a structural model against a behavioral model.

2.7.2 Semantic-based Approaches. Extraction is a clever tool for manipulating models. However, this template-based approach relies on syntactic substitutions. When applied to VHDL models, it makes no use of what the VHDL language actually means in terms of the performance of real hardware. To perform verifications between behavioral and structural models, one needs a formalization of the semantics of VHDL. These semantics need to be based on some logic to allow rigorous verification by formal proof.

Auletta devised a translation from a restricted subset of the process algebra CSP to VHDL (Auletta, 1991). In performing the translation, he strove for *synthesizable* VHDL, meaning models of finite state machines in the *register transfer logic* (*RTL*) style. One semantic mismatch he noted was that while CSP allowed the expression of non-determinism, VHDL did not. To cope with this non-determinism when translating to VHDL he used a "scheduling mechanism." This insight into how VHDL might model indeterminism is enlightening. However, for this dissertation, the *reverse* translation, *i.e.*, *from* VHDL *to* CSP (or similar algebra) is viewed as the more interesting. CCS makes a better target algebra anyway. CSP is inadequate for dealing with many concurrency issues, due to its inability to express internal action. CSP requires mutual simultaneous agreement between processes for communication to occur. Whereas problems such as deadlock occur when there is no such agreement, a deadlocking process waits forever on data that will never be sent.

Van Tassell (van Tassel, 1994) defined a formal semantics for a limited subset of VHDL using the language *Higher Order Logic* (HOL) (M. Gordon, 1987; 1992). His "nano-VHDL" is a very restricted subset of VHDL that captures the basic VHDL semantics. Using HOL, van Tassell wrote abstract syntax to formally define the semantics of a limited number of VHDL constructs. He then used the HOL proof assistant to perform symbolic simulations on the resulting HOL models. Limitations of van Tassel's work are: (1) the subset is extremely small, (2) the translation from VHDL to HOL is manual, and (3) he formalizes the simulation engine of the VHDL LRM, such that his semantics are insensitive to internal action and cannot embody the properties desired to establish safe substitution and to detect deadlock and other invariants.

Jamsik and Bickford used a logic-based approach to formalizing the semantics of VHDL (Jamsik and Bickford, 1994). This model checking approach uses a family of formal specification tools and languages referred to collectively as *Larch* (Guttag and Horning, 1993). In this approach, VHDL entities are modeled (quite naturally) in VHDL. Requirements or specifications are written in a special requirements language called a *Larch Interface Language*, or *LIL*. A different LIL is generated for each target language. VHDL-LIL statements are embedded as comments called *annotations* in the VHDL code. This results in *judgments*, which are logical statements to the effect that an entity E satisfies a requirement φ. These judgments are then proven with the aid of a set of axioms and inference rules governing judgments.

Jamsik and Bickford separate out requirements as annotations from the behavioral model, the VHDL top-level behavioral model itself. Their work expresses a general logic-based semantics not limited to a simple formalization of the simulation semantics. However, their annotations appear to be limited to properties incumbent on named signals. It is not apparent how certain concurrency properties not tied to a specific signal, such as freedom from deadlock, can be expressed, if indeed they can.

Hua and Zhang (Hua and Zhang, 1993) translated VHDL into a formal logic and used theorem proving for verification. Their tool, VAT (VHDL to Algebraic Translator), turns VHDL into RRL (Rewrite Rule Laboratory) syntax. The VAT translator maps structural VHDL into RRL axioms and maps behavioral VHDL into RRL theorems. Hence VAT creates an axiomatic system based on the hardware implementation, and the specification is thus a set of theorems to be proved about the hardware.

The VAT approach is very similar to one goal of the present research, *i.e.,* to forge a semantic link between VHDL and some logic. However, the work is limited to a "significant subset" of VHDL, and the hardware verified must be in the synchronous design style. Of course, the synchronous design style is very widely-used, but asynchronous design is more fundamental, and concurrency issues are more likely to arise in asynchronous designs. Furthermore, Hua and Zhang's "significant subset" is not a proper subset of VHDL. They invent additional VHDL syntax for the convenience of the VAT tool. They add the symbol <<= to denote connections which involve feedback, and the keyword *algebraic* to denote the expression of a requirement. Any such decoration of VHDL code before verification ought to be avoided due to the possible introduction of errors.

Read and Edwards (Read and Edwards, 1994) translated VHDL to Boyer-Moore logic. Boyer-Moore Logic is a quantifier-free first order logic with equality. Its syntax is similar to LISP. Their translation to VHDL works in two stages:

(1) VHDL syntax is mapped to Boyer-Moore expressions.

(2) "Stage 2 is an operational definition of a VHDL simulation kernel."

Stage (1) has the happy result of reducing the great number of VHDL constructs to the much smaller catalog of Boyer-Moore functions. Stage (2) creates a "formal

simulator for VHDL." These researchers too have formalized the simulation semantics, rather than expressed higher semantics.

Limitations of Read and Edwards' technique are many. First, the associated theorem prover, NQTHM, is ungainly. It is not fully automated, and must be guided. As a result, these researchers were essentially without results. They attempted one small example, but then stated "the equivalence theorem …remains unproved." They also note that their technique "loses instance names." Multiple architectures of a single entity are known by the entity name. The architecture name is thrown away. They defend this practice by stating that this lack of differentiation "maintains the association between them." This attitude seems very naive. Just because a human designer creates two architectures which he *believes* to represent the same entity does not mean they actually are equivalence, congruent, conformant or anything. This is *why* one verifies designs in the first place.

Finally, Read and Edwards treat variables like signals, that is, their tool deletes any indication of which is which. Hence when assignments are made to variables, their values are updated, not immediately, but only after the simulation clock advances. This is a serious violation of the VHDL semantics!

Examples of commercial formal verification tools for VHDL are Abstract Hardware's CheckOff-M and CheckOff-E (Musgrave and others, 1997). CheckOff-E is a formal equivalence checker, and CheckOff-M is a model checker. Both provide links from VHDL to *CIL*, a restricted form of the temporal logic *CTL* (Burch, 1989). CheckOff-M in particular will check temporal properties of behavioral, RTL, or structural VHDL models. The literature provided also claims that concurrency issues such as deadlock and race can be detected. However, the toolset is limited to the evaluation of deterministic automata. Therefore, higher equivalence semantics such as bisimulation are indistinguishable from trace equivalence.

Many modern "formal verification" tools for VHDL were displayed at the 2001 Design Automation Conference. These providers generally add the VHDL *assert* statement to models to force the gathering of statistics during simulation. The characterization of this technique as "formal verification" is a misnomer.

2.8 Hardware Order Relations and Conformances

A major goal of this dissertation is to establish a formal conformance relation that accurately captures intuitive notions of when a device adheres to a specification model, or when a part of unequal capability can be substituted for another part. This relation, to be called *congruent weak conformance*, is a partial ordering among hardware agents. This section explores other such asymmetric process ordering relations presented in the literature, known variously as *preorders*, *partial orders* and *conformances*. Rather than introduce the special notation for each, '≥' will be used as a general ordering symbol.

Arun-Kumar presents an *efficiency* preorder (Arun-Kumar and Hennessy; 1992; Arun-Kumar and Natajaran, 1995). An efficiency preorder $P \geq Q$ requires that $P \approx Q$ with P being "faster than" or "more efficient than" Q. This speed or efficiency is measured by the amount of internal computation required. In essence an efficiency preorder counts τ actions. Thus, if $P \stackrel{a}{\Rightarrow}$ by way of a direct $\stackrel{a}{\rightarrow}$ whereas $Q \stackrel{a}{\Rightarrow}$ via $\stackrel{\tau}{\rightarrow} \stackrel{a}{\rightarrow}$ then P is faster than Q. One limitation of the efficiency preorder is the assumption that all τ actions have unit weight. This rough measure of efficiency is often not realistic. Secondly, the efficiency preorder establishes an ordering within each ≈-equivalence class. There is no preorder between processes that are not observationally equivalent. Hence the efficiency preorder does not model the compliance of an implementation to a specification, where the observable behaviors can differ.

A related concept is the *divergence preorder* (Ingólfsdóttir and Schalk, 1995). A *divergence* is an unending chain of internal computation, such as in $D \stackrel{def}{=} \tau.D + \bar{a}.D$,

where τ can execute indefinitely and starve $\overline{a}.D$. For the divergence preorder, $P \geq Q$ when $P \approx Q$ but Q may diverge more than P. Like the efficiency preorder, the divergence preorder requires observational equivalence, and the desired implementation-specification relation is not modeled.

A *faster-than* preorder uses an extended CCS that associates worst-case execution times with actions (Lüttgen and Vogler, 2001). Thus agents can possess execution times resembling real operation. The faster-than preorder is again an ordering among $\approx$-equivalent agents and does not capture the desired compliance ordering among specifications and implementations.

Some researchers take note that the transition graph of a process creates an ordering among its derivatives (Godefroid, 1995; Alur and others, 1997; Corradini and others, 1997; Degano and Priami, 1999). Thus if a transition $P \overset{s}{\Rightarrow} Q$ exists then $P \geq Q$. This can be called a *causal* of *derivational* preorder. Intuitively, these processes are understood to be ordered by priority of occurrence. Again, this causal preorder does not capture the desired implementation-to-specification relationship, where $I \geq S$ should apply at *instants* of time.

Segala presents a *quiescent preorder* over processes (Segala, 1994). It compares only *quiescent* states—those that only accept inputs—and is undefined over the many intermediate states capable of output or internal action. Segala shows that the quiescent preorder *is* substitutable, and therefore a congruence. However, since only quiescent states are compared, he side-steps the issue of initial instability, which can affect congruence (Milner, 1989:112). The "greater" (left-hand) process can possess unspecified output pins. This dissertation calls such excess pins *extraneous*. The quiescent preorder handles extraneous outputs by hiding them prior to any attempt to compare agents. This simplifies the analysis, but loses the fact that the extraneous action set can change depending on which two models are compared. Finally, all required

outputs must be "yielded" by the implementation, so the quiescent pre-order does not exploit output concurrency.

Preorders have also been defined based on *testing* semantics (Hennessy, 1988:Chapter 2). A *test* is a sequence of input and output actions where the inputs become stimuli for the device under test, and the outputs denote expected responses. If the expected responses are achieved, the device or model *passes* the test. Possible non-deterministic execution is allowed via *may* and *must* testing. A process *P may* pass test *e* if *P* has an execution path that passes *e*. Other paths that fail *e* are allowed. *P must* pass *e* when there are no executions for which it would fail. The *may* and *must* preorders are defined in terms of test set containment. Hence $P \leq_{may} Q$ if $\{e : P$ may satisfy $e\} \subseteq \{e : Q$ may satisfy $e\}$ with a similar definition for $\leq_{must}$.

Consider whether testing preorders can be used to express compliance. Let $S \overset{def}{=} a.(\overline{o}.\overline{p} + \overline{p}.\overline{o})$. *S* has an output concurrency permitting $\overline{o}$ and $\overline{p}$ to occur in either order. Let $I \overset{def}{=} a.\overline{o}.\overline{p}$. *I* complies with *S* by having selected one output interleaving. The set of tests that *S may* pass is $\{a.\overline{o}.\overline{p}, a.\overline{p}.\overline{o}\}$. For *I*, that set is $\{a.\overline{o}.\overline{p}\}$, so $I \leq_{may} S$. The set of tests that *S must* pass is empty, whereas the *must* pass set for *I* is $\{a.\overline{o}.\overline{p}\}$. Hence $S \leq_{must} I$. One might suppose that $I \leq_{may} S \leq_{must} I$ denotes the proper compliance relationship. Yet $NIL \leq_{may} S \leq_{must} NIL$ and NIL is *not* compliant to *S*. One concludes that the testing preorders, as defined, do not support the expression of compliance.

Conformances are asymmetrical relations with the specification appearing on the right and the implementation on the left. Stevens studied conformances and developed a new property called *logic conformance* (Stevens, 1994:136-44).

Definition 2-6. (Stevens, 1994: Definition 30). Implementation *I logically conforms* to specification *S*, written $I \succeq_l S$, iff $\forall \alpha \in \mathcal{A}ct, \forall \beta \in \overline{\mathcal{A}} \cup \{\tau\}$ and $\forall \gamma \in \mathcal{A}$

   (1) Whenever $S \xrightarrow{\alpha} S'$ then for some $I' : I \xRightarrow{\hat{\alpha}} I'$ and $I' \succeq_l S'$

   (2) Whenever $I \xrightarrow{\beta} I'$ then for some $S' : S \xRightarrow{\hat{\beta}} S'$ and $I' \succeq_l S'$

   (3) Whenever $I \xrightarrow{\gamma} I'$ and $S \xRightarrow{\gamma}$ then for some $S' : S \xRightarrow{\gamma} S'$ and $I' \succeq_l S'$

Logic conformance respects preemptive internal actions and abstract away all others. It is sensitive to the branching structure of agents. It detects deadlock, and requires that deadlocks in the implementation must match deadlocks in the specification. Part (1) contains the basic demand that all specified behaviors be implemented. Part (2) demands that every implemented output correspond to a specified output event. In part (3), the additional premise $S \xRightarrow{\gamma}$ allows the implementation to accept unspecified inputs.

     Conformances treat input and output distinctly. Hence, for conformance relations, the overbar is identified specifically with output. This departs from previous usage, where the overbar is used merely for synchronization, and the association with either input or output is arbitrary. The association of the overbar strictly with output is enforced throughout the remainder of this dissertation.

     Logic conformance has shortcomings that need remedy. First of all, part (1) is overly restrictive with respect to outputs. It requires *I* to implement *every* specified output action, even when there is output concurrency. Secondly, part (2) makes no allowance for the implementation to generate output signals outside of the specification.

2.9 Summary

     This chapter discussed various verification methods, introduced the concept of *process algebra*, and outlined the process algebra CCS. CCS was then used as a tool to discuss the differentiation of various hardware equivalences, of which four were

presented. Modal and temporal logics were presented as a means to assert requirements on hardware models written in some process algebra.

Section 2.7 then considered how formal verification methods have been applied in the past to VHDL models. Extraction techniques, which are purely syntactic, were first presented, followed by several semantic-based approaches. Limitations of past techniques include:

(1) Inability to compare structure to behavior.

(2) Formalization of the VHDL simulation semantics only.

(3) Severely limited VHDL subset.

(4) Artificially created additional VHDL syntax.

(5) Requirement to manually edit or annotate VHDL code prior to verification.

(6) Oversimplification of the semantics. (For example, treating variables and signals as the same.)

Section 2.8 presented various ordering relationships that are potential competitors to congruent weak conformance. Limitations of these techniques include:

(1) The ordering is not based on compliance, but some other measure.

(2) The ordering is applied within $\approx$-equivalence classes only.

(3) The ordering is not defined for all states.

(4) The ordering does not allow output concurrency options.

Subsequent chapters will describe the output of the present research which seeks to alleviate some of the above limitations.

III. Weak Conformations

This chapter develops the precursor relations called *weak conformations*. First, a simple example is given, using the representational power of CCS to exhibit a specification and a compliant implementation.   The example yields intuition from which the four transitional laws governing weak conformations are derived. Extensive formal results are derived for these precursor properties.

3.1 Compliance Example

Consider a circuit specified to convert binary-coded-decimal (BCD) to pure decimal. It takes four bits to encode a decimal digit, so the converter will have four inputs, one for each of the encoding bits. Call the inputs *a, b, c* and *d*. The ten outputs will be labeled $\bar{o}_0, \bar{o}_1, \ldots, \bar{o}_9$, one for each decimal digit detected. One can think of the outputs as ten indicator lights. In a CCS model of the specification, each time there is change on an input bit, one of the output lights turns on and another is extinguished. Since CCS models transitions and not level signals, there will be two output transitions concurrently, but it will not be readily apparent which is turning on and which is turning off. Assume the parent system does not care if *momentarily* two are lit, or none.   The specification will allow either.   The specification model will have ten named states corresponding to each decimal digit.[1] The specification model is identified with the root state, corresponding to decimal zero:

---

[1]   It is convenient to name 10 of the states, but the model has many more intermediate states. There is a state after the occurrence of each atomic action.

$$S \stackrel{def}{=} S0 \stackrel{def}{=} a.(\ \bar{o}_0.\bar{o}_1.S1 + \bar{o}_1.\bar{o}_0.S1)$$

$$+\ b.(\ \bar{o}_0.\bar{o}_2.S2 + \bar{o}_2.\bar{o}_0.S2)$$

$$+\ c.(\ \bar{o}_0.\bar{o}_4.S4 + \bar{o}_4.\bar{o}_0.S4)$$

$$+\ d.(\ \bar{o}_0.\bar{o}_8.S8 + \bar{o}_8.\bar{o}_0.S8) \tag{3-1}$$

Similar definitions exist for states *S1* through *S9*. However the code is ungainly because two terms have to be presented each time there is concurrency on two outputs.   The shorthand notation $(\bar{o}_0 \mid \bar{o}_1)$ can now be used to express the concurrency of output signals while economizing on the code.[2]

$$S0 \stackrel{def}{=} a.(\ \bar{o}_0 \mid \bar{o}_1).S1 + b.(\ \bar{o}_0 \mid \bar{o}_2).S2 + c.(\ \bar{o}_0 \mid \bar{o}_4).S4 + d.(\ \bar{o}_0 \mid \bar{o}_8).S8$$

$$S1 \stackrel{def}{=} a.(\ \bar{o}_1 \mid \bar{o}_0).S0 + b.(\ \bar{o}_1 \mid \bar{o}_3).S3 + c.(\ \bar{o}_1 \mid \bar{o}_5).S5 + d.(\ \bar{o}_1 \mid \bar{o}_9).S9$$

$$S2 \stackrel{def}{=} a.(\ \bar{o}_2 \mid \bar{o}_3).S3 + b.(\ \bar{o}_2 \mid \bar{o}_0).S0 + c.(\ \bar{o}_2 \mid\ \bar{o}_6).S6$$

$$S3 \stackrel{def}{=} a.(\ \bar{o}_3 \mid \bar{o}_2).S2 + b.(\ \bar{o}_3 \mid \bar{o}_1).S1 + c.(\ \bar{o}_3 \mid \bar{o}_7).S7$$

$$S4 \stackrel{def}{=} a.(\ \bar{o}_4 \mid \bar{o}_5).S5 + b.(\ \bar{o}_4 \mid \bar{o}_6).S6 + c.(\ \bar{o}_4 \mid \bar{o}_0).S0$$

$$S5 \stackrel{def}{=} a.(\ \bar{o}_5 \mid \bar{o}_4).S4 + b.(\ \bar{o}_5 \mid \bar{o}_7).S7 + c.(\ \bar{o}_5 \mid \bar{o}_1).S1$$

$$S6 \stackrel{def}{=} a.(\ \bar{o}_6 \mid \bar{o}_7).S7 + b.(\ \bar{o}_6 \mid \bar{o}_4).S4 + c.(\ \bar{o}_6 \mid \bar{o}_2).S2$$

$$S7 \stackrel{def}{=} a.(\ \bar{o}_7 \mid \bar{o}_6).S6 + b.(\ \bar{o}_7 \mid \bar{o}_5).S5 + c.(\ \bar{o}_7 \mid \bar{o}_3).S3$$

$$S8 \stackrel{def}{=} a.(\ \bar{o}_8 \mid \bar{o}_9).S9 + d.(\ \bar{o}_8 \mid \bar{o}_0).S0$$

$$S9 \stackrel{def}{=} a.(\ \bar{o}_9 \mid \bar{o}_8).S8 + d.(\ \bar{o}_9 \mid \bar{o}_1).S1 \tag{3-2}$$

Only states *S0* and *S1* respond to all four inputs because combinations above **1001** are illegal under the BCD code. Omitting these transitions in *S2* to *S9* constitutes the specification's guarantee that the illegal input combinations will not be received.

---

[2] This shorthand, which one can think of as a "parallelism of actions," is not part of the CCS formal syntax.

Given the above specification *S*, what sort of circuit would make a conforming implementation?   A 4:16 demux, as shown in Figure 3-1, is an obvious choice.   The inputs *a, b, c*, and *d* form the four select lines of the demux. Of the 16 outputs, only 10 are used, and six are left unconnected. A fifth input pin represents the multiplexed input. In this application that pin is tied to '1'. Note therefore that a compliant implementation must have a pin for every I/O pin called out by the specification, though it may have more.
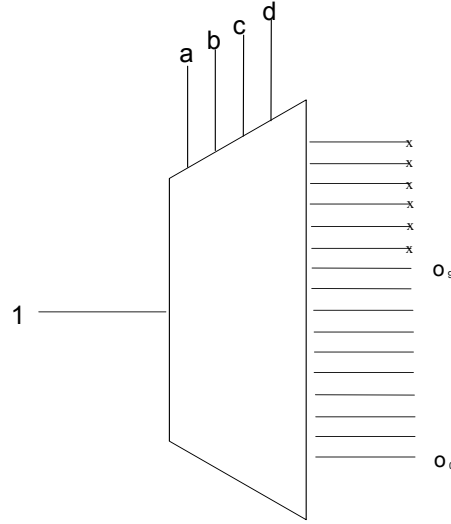
Figure 3-1. 4:16 Demux

A "first cut" CCS model for this demux could read just like the specification model but with the missing input transitions added and the extra outputs generated.

$$I \stackrel{def}{=} I0 \stackrel{def}{=} a.(\ \bar{o}_0 \mid \bar{o}_1).I1 \ + b.(\ \bar{o}_0 \mid \bar{o}_2).I2 \ \ + c.(\ \bar{o}_0 \mid \bar{o}_4).I4 \ \ + d.(\ \bar{o}_0 \mid \bar{o}_8).I8$$

$$I1 \stackrel{def}{=} a.(\ \bar{o}_1 \mid \bar{o}_0).I \ \ + b.(\ \bar{o}_1 \mid \bar{o}_3).I3 \ \ + c.(\ \bar{o}_1 \mid \bar{o}_5).I5 \ \ + d.(\ \bar{o}_1 \mid \bar{o}_9).I9$$

$$\dots$$

$$I9 \stackrel{def}{=} a.(\ \bar{o}_9 \mid \bar{o}_8).I8 \ \ + b.(\ \bar{o}_9 \mid \bar{o}_{11}).I11 + c.(\ \bar{o}_9 \mid \bar{o}_{13})\ I13 + d.(\ \bar{o}_9 \mid \bar{o}_1).I1 \qquad (3\text{-}3)$$

This implementation has more states than the specification due to its ability to execute illegal sequences. Indeed, this is allowable since the specification guarantees that these additional states are unreachable. One might hastily conclude that implementations must duplicate *all* the states of the specification, with additional states allowed. Yet this is not the case. Though the example implementation *I* gratuitously generates *all* the possible output interleavings allowed by *S*, in reality it would be both difficult and counterproductive to create such a device. A real, physical layout results in finite delays along various paths. Most likely, the same interleaving appears every time in a physical implementation, especially when the delays are due solely to passive components.

Consider a diagram of the transitions from *S1* to *S2* (Figure 3-2). Concurrency of outputs is represented by a characteristic diamond shape. Clearly, the implementation need only navigate one path through this diamond, or through any such output "burst." The same is not true for inputs. When an input concurrency is present, as in the case of the *C* element (Equation 2-1), the implementation must remain poised to accept any possible interleaving that may come and therefore must be able navigate all paths through a specified input burst.

S1

a

'o2   'o1

'o1   'o2

S2
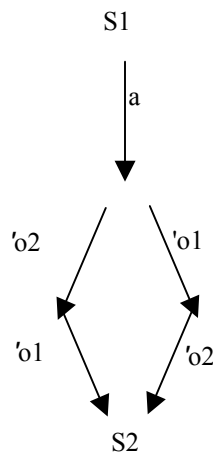
Figure 3-2. Output Concurrency Diamond

To exploit this allowance to chose among output interleavings, a "second cut" implementation J chooses specific output interleavings where possible. This implementation might look something like this:

$$J \stackrel{def}{=} J0 \stackrel{def}{=} a.\bar{o}_0.\bar{o}_1.J1 + b.\bar{o}_0.\bar{o}_2.J2 + c.\bar{o}_4.\bar{o}_0.J4 + d.\bar{o}_0.\bar{o}_8.J8$$

$$\ldots \tag{3-5}$$

and so forth. One specific interleaving is chosen at each output concurrency.

Thus when presented with an output concurrency, the implementation can implement *any* or *all* the paths, as long as at least *one* path is implemented. If one considers the possible paths to form a set, then the implementation must select some non-empty subset. This idea will be captured later by the notion of *maxoctset*.

Consider now the question of *behaviors* or *sequences* of actions. *I* and *J do* accept more input behaviors than *S* specifies due to their ability to decode illegal, non-BCD inputs. However, they could be faulty for codes '11' through '15' and still function as BCD converters. These behaviors are irrelevant. Designers will exploit this "don't care" region of behavior to produce more efficient designs.

In this example neither *S* nor its implementations *I* and *J* contain internal action τ. These models are strictly behavioral, and hidden actions usually arise in structural models, where there can be communication between internal signals. Unlike *I* and *J*, most implementation models, in practice, will be structural; and a structural model with no internal communication is a rarity. Therefore, τ actions in the implementation are virtually inevitable.

Furthermore, τ actions in the specification are likely as well. Some practitioners advocate that complex specification models be presented structurally (Stevens and others, 1993). For complex behavior, a purely "flat" specification model will have an overwhelming number of states. Breaking the model into a few parallel models can

greatly simplify the expression of the specification, though they can introduce τ in the specification.

The BCD decoder example shows how a compliant implementation can exceed the specification in the number of I/O pins, and can also generate illegal behavior in the unreachable state space. In general the implementation can possess *more* behaviors than the specification, though it can get by with *fewer* output behaviors. In the next section, the intuition derived from this example will be developed formally, yielding a set of properties called *weak conformations*. Weak conformations are precursor properties to the target relation to be called *congruent weak conformance*.

3.2 Notation

To transform intuitive ideas on compliance into formal properties, additional symbolism is needed.

First of all, the notion of *sort* used here differs from Milner's usage. Milner uses *syntactic* sorts where here *semantic* sorts are more useful. To derive a syntactic sort, one simply catalogs the symbols appearing in the expression of an agent and its derivatives. Some of these symbols may in fact be unreachable from the root state. Since they will never be encountered, they are excluded from the *semantic* sort. The efficiency of deriving semantic sorts is not an immediate concern since the initial intended use of congruent weak conformance does not require the actual generation of sorts by an automated tool.

Thus $\mathcal{L}(P)$ denotes the visible *semantic sort* of P. Similarly, $\mathcal{A}(P)$ and $\overline{\mathcal{A}}(P)$ are the *semantic input* and *output* sorts of P, respectively, with $\mathcal{A}ct(P)$ being the *semantic action set*. Note that $\mathcal{A}(P) \cup \overline{\mathcal{A}}(P) = \mathcal{L}(P) \subseteq \mathcal{A}ct(P)$ and that although $\mathcal{A}ct(P)$ *may* include τ, $\mathcal{L}(P)$, $\mathcal{A}(P)$ and $\overline{\mathcal{A}}(P)$ never do.

The forward slash '/' denotes "excess of…over…" for strings (Milner, 1989:Definition 11.6). Informally, *r/s* is the string *r* where the symbols it shares with *s*

have been removed. This removal occurs from left to right and takes note of the multiplicity of symbols within *s*. Thus, if the symbol *a* appears twice within *s* then no more than two occurrences of *a* are removed from *r*. As examples: *a.b.c/a.c = b, a.b.a.b/a = b.a.b,* and *a.b.c/a.a = b.c.*

'↾' denotes the projection operation and normally applies to the projection of an action string onto a set. Thus $t ↾ \mathcal{A}(S)$ is the string *t* with all actions removed *except* those in $\mathcal{A}(S)$.

A new notation denotes the additional pins of an implementation that exceed those of the specification. These are called *extraneous* pins. $\overline{\mathcal{E}xtr}\,(I,S)$ is the set of extraneous output of *I* with respect to *S*, and $\mathcal{E}xtr(I,S)$ the extraneous input set.

Definition 3-1. Let *I* and *S* be process agents:

(1) $\mathcal{E}xtr(I,S) \equiv \mathcal{A}(I) - \mathcal{A}(S)$ is the *extraneous input sort* of *I* with respect to *S*.

(2) $\overline{\mathcal{E}xtr}\,(I,S) \equiv \overline{\mathcal{A}}(I) - \overline{\mathcal{A}}(S)$ is the *extraneous output sort* of *I* with respect to *S*.

3.3 Weak Confluence and Maxoctsets

Weak conformations use the notion of *confluence*, a restricted form of determinism. There are both *strong* and *weak* versions of confluence, with weak being the more interesting. One of Milner's results will serve as a working definition of weak confluence (Milner 1989, Proposition 11.11). Shown diagrammatically,

Definition 3-2. If *P* is *weakly confluent* then

$$
\begin{array}{ccc}
 & r & \\
P & \Rightarrow & P' \\
s \Downarrow & & \Downarrow s/r \\
P'' & \Rightarrow & \approx \\
 & r/s &
\end{array}
$$

The diagram is interpreted such that the top and left transitions imply the bottom and right transitions. The anonymous successors of P' and P'' are weakly bisimilar and denoted with '$\approx$', that being the largest weak bisimulation that all weakly bisimilar states must enjoy. In arriving at the lower right via different paths, the same visible actions are encountered the same number of times, albeit the order of the actions may be different. No visible action is preempted from occurring its appointed number of times, and the strings $r.s/r$ and $s.r/s$, though different, have the same net effect. Strings such as these, which are equivalent up to permutation, are called *confluence equivalent.*

Definition 3-3. $\forall r,s \in \mathcal{L}^*$, r and s are *confluence equivalent* sequences, written $r =_{\text{conf}} s$, if $r/s = \varepsilon = s/r$.

In the presence of confluence, such sequences always terminate at the same state up to $\approx$.

A new property called *local confluence* applies to agents wherein isolated portions of their transition graphs can exhibit confluence, even if the root agent does not. Milner's confluence is a global property, insisting that all exiting sequences preserve the confluence. Local confluence, by contrast, is content with a portion of the transition graph that resembles the confluence diagram. Other transitions which destroy global confluence are ignored.

Definition 3-4. Let $s \in \mathcal{L}^*$. Agent P is *locally confluent* with respect to s if $P \overset{s}{\Rightarrow}$ and $\forall r =_{\text{conf}} s$, whenever $P \overset{s}{\Rightarrow} P'$ and $P \overset{r}{\Rightarrow} P''$ then $P' \approx P''$.

When local confluence occurs, all exiting confluence-equivalent sequences terminate at states within the same $\approx$-equivalence. One often gives these $\approx$-equivalent states anonymity and writes $P \overset{r}{\Rightarrow} \approx P'$ for all such r. Note that P can be said to be locally confluent with respect to *any* of the sequences r. All such sequences form a set:

Definition 3-5. Let P be locally confluent with respect to s. The set $\{\ r =_{conf} s : P \overset{r}{\Rightarrow}\ \}$ is the *confluent transition set* (*CT set*) of P with respect to s.

The CT set does not contain *all* permutations of s. It contains only those of which P is capable. Though local confluence applies to both inputs and outputs, it is the local confluence among *outputs* that can be exploited by an implementation. CT sets composed only of outputs are called *octsets*.

Definition 3-6. Let X be a CT set of P with respect to s. X is an *output confluent transition set* (*octset*) of P with respect to s if $s \in \overline{A}^{\,+}$.

For octsets, member sequences must be of non-zero length ($s \in \overline{A}^{\,+}$). This avoids the burden of a trivial octset $\{\varepsilon\}$ which lends no flexibility in design anyway. In fact, since only a single member sequence needs to be implemented, the desire will be to have large octsets, composed of lengthy sequences, for these will give the greatest flexibility in design. Thus, the "lengthiest" octsets that can be built are called *maxoctsets*.

Definition 3-7. Let X be an octset of P with respect to s. X is a *maxoctset* of P if $\nexists t \in \overline{A}(P)^{+}$ such that P has an octset with respect to st.

In the extreme case, when no flexibility in design is offered, a maxoctset is a singleton set whose lone output sequence *must* be implemented. *Every* output transition $\overset{\bar{a}}{\rightarrow}$ participates in *some* maxoctset, though it may be as trivial as $\{a\}$. Thus the laws governing the implementation of output can be defined over the maxoctsets of a specification, and not over the individual output actions themselves.

3.4 Weak Conformations

A family of properties called *weak conformations* is now introduced. Of these, *weak conformance* will be defined later as the largest weak conformation. Weak conformations are asymmetric relations with the specification agent on the right and the implementation on the left.

Definition 3-8. A binary relation on processes, $\mathcal{W} \subseteq \mathcal{P} \times \mathcal{P}$, is a *weak conformation* if $\forall \alpha \in \mathcal{A}(S) \cup \{\tau\}$, $\forall \beta \in \overline{\mathcal{A}}(I) \cup \{\tau\}$, $\forall \gamma \in \mathcal{A}(S)$, $I \, \mathcal{W} \, S$ *implies* the following four laws:

Law of Specified Input or Tau (LSIT)

Whenever $S \overset{\alpha}{\leadsto} S'$ then $\exists t \in (\mathcal{A}(S) \cup \overline{\mathcal{E}xtr}(I,S))^*$ such that

    (1) $I \overset{}{\Rightarrow} I'$

    (2) $t \!\restriction\! \mathcal{A}(S) = \hat{\alpha}$

    (3) $I' \, \mathcal{W} \, S'$

Law of Specified Output (LSO)

Let X be a maxoctset of S. $\exists s \in X$ and $\exists t \in \overline{\mathcal{A}}(I)^+$ such that

    (1) $S \overset{s}{\Rightarrow} S'$

    (2) $I \overset{t}{\Rightarrow} I'$

    (3) $t \!\restriction\! \overline{\mathcal{A}}(S) = s$

    (4) $I' \, \mathcal{W} \, S'$

Law of Implemented Input (LII)

Whenever $I \overset{\gamma}{\leadsto} I'$ and $S \overset{\gamma}{\Rightarrow}$ then

    (1) $S \overset{\gamma}{\Rightarrow} S'$

    (2) $I' \, \mathcal{W} \, S'$

Law of Implemented Output or Tau (LIOT)

Whenever $I \overset{\beta}{\Rightarrow} I'$ and $\delta \equiv \beta \restriction \overline{\mathcal{A}}(S)$ then

(1) $S \overset{\delta}{\Rightarrow} S'$

(2) $I' \, \mathcal{W} \, S'$

LSIT describes the obligation of the implementation when the specification requires an input or τ. The implementation answers by performing a string t.   Both agents then evolve to derivatives I' and S' that also share the relation $\mathcal{W}$. String t contains one occurrence of an input when α is visible and none when $\alpha = \tau$. The remainder of t contains extraneous outputs that can occur without harm because they are unknown to the specification.   In the statement of LSIT, these extraneous outputs are filtered by the projection onto $\mathcal{A}(S)$. Other than a lone $\hat{\alpha}$, t can contain no other inputs. Even those inputs in $\mathcal{E}xtr(I,S)$ are prohibited. If t *did* contain such unspecified input actions, the implementation would wait forever on those inputs, and would thus be blocked.

LSO describes how an implementation answers specified output activity.   At least *one* sequence s, though possibly more, from each maxoctset must be "matched" by the implementation. The implementation matches s with t. String t contains *all* the actions of s, in the same sequence that they appear in s. As before, t can further incorporate any number of extraneous outputs without harm. One will often say that t *implements s,* or alternately, that t *implements X*, since s is the representative of the entire maxoctset X.

LII is a reuse of Definition 2-6 (2). It addresses the care that must be taken when the implementation performs an input action within the specification sort. If the specification is not *immediately* capable of such action, there is no harm done because that action will not be forthcoming anyway. If the specification *is* immediately capable $(S \overset{\gamma}{\Rightarrow})$ then of course the implementation must match that action in accordance with LSIT. However, LII goes beyond that to state that I is prohibited from any other use of

specified input symbols, except those arising by LSIT. Otherwise, the implementation could stray into illegal behavior triggered by a legal input.

LIOT addresses the occurrence of taus and specified outputs in the implementation. Although the implementation can freely engage in *extraneous* outputs, LIOT requires that any use of *specified* output symbols in the implementation be limited to those that arise by legal application of LSO. This prevents the implementation from issuing illegal behavior at pins that are observed by the containing system.

Weak conformations are "weak" because, like weak bisimulations, they abstract away τ actions. For completeness, a definition of *strong conformation* appears in Appendix A, but no attempt is made to develop strong conformation theory or to pursue it toward a *congruent strong conformance* relation. As always, it is the weak case that is more interesting and useful, and merits further development.

For convenience in executing proofs, corollary laws to the weak conformation definition can be derived:

Corollary 3-1. Whenever $I \, \mathcal{W} \, S$ for weak conformation $\mathcal{W}$, the following laws hold:

Law of Input Coverage (LIC). $\mathcal{A}(S) \subseteq \mathcal{A}(I)$

Law of Output Coverage (LOC). $\overline{\mathcal{A}}(S) \subseteq \overline{\mathcal{A}}(I)$

Law of Specified Epsilon (LSE). *Whenever $S \Rightarrow S'$ then $\exists t \in \overline{\mathcal{E}xtr}\,(I,S)$* such that*

    (1) $I \stackrel{t}{\Rightarrow} I'$

    (2) $I' \, \mathcal{W} \, S'$

Law of Specified Abstracted Input (LSAI).
$\forall \alpha \in \mathcal{A}(S)$: *whenever $S \stackrel{\alpha}{\Rightarrow} S'$ then $\exists t \in (\mathcal{A}(S) \cup \overline{\mathcal{E}xtr}\,(I,S))^{+}$ such that*

    (1) $I \stackrel{t}{\Rightarrow} I'$

    (2) $t \upharpoonright \mathcal{A}(S) = \alpha$

(3) $I' \mathcal{W} S'$

Law of Specified Input Strings (LSIS).

$\forall s \in \mathcal{A}(S)^+$: *whenever* $S \overset{s}{\Rightarrow} S'$ *then* $\exists t \in (\mathcal{A}(S) \cup \overline{\mathcal{E}xtr}(I,S))^+$ *such that*

(1) $I \overset{t}{\Rightarrow} I'$

(2) $t \restriction \mathcal{A}(S) = s$

(3) $I' \mathcal{W} S'$

Law of Implemented Epsilon (LIE). *Whenever* $I \Rightarrow I'$ *then*

(1) $S \Rightarrow S'$

(2) $I' \mathcal{W} S'$

Law of Implemented Abstracted Input (LIAI). $\forall \gamma \in \mathcal{A}(I)$: *whenever* $I \overset{\gamma}{\Rightarrow} I'$ *then*

(1) $S \overset{\gamma}{\Rightarrow} S'$

(2) $I' \mathcal{W} S'$

Law of Implemented Input Strings (LIIS). $\forall s \in \mathcal{A}(S)^+$: *whenever* $I \overset{s}{\Rightarrow} I'$ *and* $S \overset{s}{\Rightarrow}$ *then*

(1) $S \overset{s}{\Rightarrow} S'$

(2) $I' \mathcal{W} S'$

Law of Implemented Abstracted Output (LIAO). $\forall \beta \in \overline{\mathcal{A}}(I)$: *whenever* $I \overset{\beta}{\Rightarrow} I'$ *and* $\delta \equiv \beta \uparrow \overline{\mathcal{A}}(S)$ *then*

(1) $S \overset{\delta}{\Rightarrow} S'$

(2) $I' \mathcal{W} S'$

Law of Implemented Output Strings (LIOS). $\forall s \in \overline{\mathcal{A}}(I)^+$: *whenever* $I \overset{s}{\Rightarrow} I'$ *then*

(1) $S \overset{s}{\Rightarrow} S'$

(2) $I' \mathcal{W} S'$

Proof: LIC and LOC follow directly from LSIT and LSO. LSE, LIE, LSIS, LIIS and LIAO yield to induction. LSAI, LIAI and LIAO are shown by replacing '$\overset{x}{\Rightarrow}$' with '$\Rightarrow \overset{x}{\rightarrow} \Rightarrow$'. $\qquad \square$

The next two propositions follow readily from Definition 3-8:

Proposition 3-2. *The process identity relation Idp is a weak conformation.*

Proof: Substitute *P* for both *I* and *S,* and *P′* for both *I′* and *S′.* $\qquad \square$

Proposition 3-3. *The union of weak conformations is a weak conformation.*

Proof: $\mathcal{V} \cup \mathcal{W}$ satisfies each law on the strength of $\mathcal{V}$ or $\mathcal{W}$ acting alone. $\qquad \square$

Milner developed *observational equivalence* $\approx$ as the largest of the *weak bisimulations*, and then strengthened it to a congruence by requiring initially stable agents (Milner, 1989:112). Since weak conformations are based on bisimulation semantics, this dissertation seeks to do the same, *i.e.,* to identify the largest weak conformation and then strengthen it to a congruence over CCS. The necessary proofs that follow make frequent use of the composition ∘ of weak conformations, relying on such compositions to also be weak conformations.[3] Hence, the demonstration that ∘ preserves weak conformation is essential. This requires that the preservation of each weak conformation law must be shown in turn.

To show the preservation of LSO, a critical result concerns the string *t* that implements a specified maxoctset. String *t* must in turn define a maxoctset in the implementation. Assurance is needed that *t* is neither lost within some maxoctset that

---

[3] ∘ will be called *relational composition* to distinguish it from the Parallel Composition of CCS processes.

exceeds it, nor that t outspans maxoctsets in the implementation due to a premature interruption of confluence. Lemma 3-4 assures the former, and 3-5 the latter.

Lemma 3-4. *Let $I \mathcal{W} S$ for some weak conformation $\mathcal{W}$, and let X be a maxoctset of S. Let t be an implementation, by I, of $s \in X$. There is* no *maxoctset Y of I with respect to some $t.t'$ unless $t' \upharpoonright \overline{\mathcal{A}}(S) = \varepsilon$.*

Proof: By contradiction.

- *Trial Hypothesis.* Assume $t' \upharpoonright \overline{\mathcal{A}}(S) = s' \neq \varepsilon$.

- By LSO, $\exists I', S'$ such that

$$S \overset{s}{\Rightarrow} S', \ I \overset{t}{\Rightarrow} I', \ t \upharpoonright \overline{\mathcal{A}}(S) = s, \ I' \mathcal{W} S'.$$

- Since $t.t' \in Y$, then $I \overset{t}{\Rightarrow} I' \overset{t'}{\Rightarrow} \approx I''$.

- $\forall y \in Y, I \overset{y}{\Rightarrow} \approx I''$ and $y =_{\text{conf}} t.y'$ by the definition of "octset."

- Since $I' \mathcal{W} S'$ then LIO demands that

$$S' \overset{s'}{\Rightarrow} S'', \ I'' \mathcal{W} S''.$$

- However, $\forall x \in X, \ S \overset{x}{\Rightarrow} S' \overset{s'}{\Rightarrow} S''$.

- $\therefore \{x.s' : x \in X\}$ is an octset of S, and X cannot be a maxoctset.

$\Rightarrow\!\Leftarrow$

Lemma 3-5. *Under the same assumptions as Lemma 3-4, there is* no *maxoctset Y of I with respect to some proper prefix t' of t (that is, $t = t'.t''$ with $t' \upharpoonright \overline{\mathcal{A}}(S) = s' \neq \varepsilon$) unless $t'' \upharpoonright \overline{\mathcal{A}}(S) = \varepsilon$.*

Proof: By contradiction.

- *Trial Hypothesis.* Assume $t'' \upharpoonright \overline{\mathcal{A}}(S) = s'' \neq \varepsilon$.

- By LSO, $\exists I', I'', S', S''$ such that

$$I \overset{t'}{\Rightarrow} I' \overset{t''}{\Rightarrow} I'', \ S \overset{s'}{\Rightarrow} S' \overset{s''}{\Rightarrow} S'', \ t'.t'' \upharpoonright \overline{\mathcal{A}}(S) = s'.s'' = s, \ I'' \mathcal{W} S''.$$

- $\forall y \in Y, I \overset{y}{\Rightarrow} \approx I'$ with $y =_{\text{conf}} t'$ by the definition of "octset."

- Since $I' \overset{t''}{\Rightarrow} I''$ then $\forall y, I \overset{y}{\Rightarrow} \approx I' \overset{t''}{\Rightarrow} I''$.

- $\therefore \{y.t'' : y \in Y\}$ is an octset, and Y cannot be a maxoctset.

$\Rightarrow \Leftarrow$

Proposition 3-6. *For weak conformations, an implementing string for a maxoctset of the specification defines a maxoctset in the implementation.*

Proof: Lemmas 3-4 and 3-5. ◻

Proposition 3-7. *Relational composition preserves LSO.*

Proof:

- Let $P \mathcal{V} Q \mathcal{W} R$ and let X be a maxoctset of R.

- By LSO $\exists s \in X$ such that $R \overset{s}{\Rightarrow} R'$ and $Q \overset{t}{\Rightarrow} Q' \mathcal{W} R'$ for appropriate t.

- By Proposition 3-6, Q has a maxoctset Y with respect to t. P must implement Y, though it may indeed not implement t itself but some other $y =_{\text{conf}} t$.

- Thus $P \overset{u}{\Rightarrow} P'' \mathcal{V} Q''$ where $y = u \restriction \overline{\mathcal{A}}(Q)$ and $Q \overset{y}{\Rightarrow} Q'' \approx Q'$. Hence $P \overset{u}{\Rightarrow} P'' \approx \mathcal{W} R'$. This is *almost*, but not quite, the desired derivative relationship.

- However, with respect to $Q \mathcal{W} R$, y is an implemented output string, so LIOS applies and $R \overset{x}{\Rightarrow} R''$ where $y \restriction \overline{\mathcal{A}}(R) = x$ and $Q'' \mathcal{W} R''$.

- Since $y =_{\text{conf}} t$ then its projection is $x =_{\text{conf}} s$. Hence, $x \in X$ and $P \overset{u}{\Rightarrow} P'' \mathcal{W} \mathcal{V} R''$. P has implemented some $x \in X$, as desired.
  ◻

Proposition 3-8. *Relational composition preserves LIOT.*

Proof: For $P \mathrel{\mathcal{V}} Q \mathrel{\mathcal{W}} R$ let $P \xrightarrow{\beta} P'$ where $\beta \in \overline{\mathcal{A}}(P) \cup \{\tau\}$.

- If $\beta = \tau$ then $P \xrightarrow{\tau} P'$ and by LIOT, $Q \Rightarrow Q'$ with $P' \mathrel{\mathcal{V}} Q'$. Applying LIE to $Q \Rightarrow Q'$ yields $R \Rightarrow R'$ with $Q' \mathrel{\mathcal{W}} R'$. Hence $P' \mathrel{\mathcal{VW}} R'$.

- If $\beta \in \overline{\mathcal{A}}(P)$ then applying LIOT to $P \mathrel{\mathcal{V}} Q$ yields

$$Q \xrightarrow{\delta} Q',\ \delta = \beta \upharpoonright \overline{\mathcal{A}}(Q),\ P' \mathrel{\mathcal{V}} Q'.$$

- There are two cases for δ: (1) $\delta = \varepsilon$ and (2) $\delta = \beta$.

- *Case 1.* Apply LIE to $Q \mathrel{\mathcal{W}} R$ yielding $R \Rightarrow R'$, $Q' \mathrel{\mathcal{W}} R'$ and hence $P' \mathrel{\mathcal{VW}} R'$.

- *Case 2.* Apply LIAO to $Q \mathrel{\mathcal{W}} R$ yielding $R \xrightarrow{\delta'} R'$ where $\delta' = \beta \upharpoonright \overline{\mathcal{A}}(R)$, $Q' \mathrel{\mathcal{W}} R'$ and hence $P' \mathrel{\mathcal{VW}} R'$.

◻

Proposition 3-9. *Relational composition preserves LSIT.*

Proof: See Appendix B.

The logical next step is to prove that the remaining law, LII, is preserved by relational composition. Unfortunately, the preservation of LII cannot be proven under the present assumptions, and the reason harks back to the instability issue faced by $\approx$. The observational congruence property $=$ solved this nicely by identifying the role of unguarded τ actions and requiring such actions to be matched in the initial agents. Milner then showed that when $P \approx Q$ and both were stable, then $P = Q$ follows (Milner, 1989:Proposition 7.10).

Within the context of weak conformations, however, the issue of instability is more complex. The preservation of LII across $P \mathrel{\mathcal{V}} Q \mathrel{\mathcal{W}} R$ fails because there is no obligation for the middle agent Q to perform an immediate $\xrightarrow{\gamma}$. LSO permits Q to perform extraneous outputs first. From the standpoint of the specification R, such outputs

are just as spontaneous and uncontrollable as τ actions. Thus, an output $\bar{x}$ that is extraneous to both A and B creates an instability in $\bar{x}.A + B$. The spontaneous occurrence of $\bar{x}$ can preempt the Choice of B. Hence *no* weak conformation exists between $\bar{x}.A + B$ and $A + B$. An otherwise stable implementation can be *relatively unstable* with respect to the specification when an extraneous output plays the role of τ. The *relative tau* symbol τ_S will denote such actions where the subscript S is the specification agent's name. Thus τ_S will admit both literal τ actions as well as output actions beyond the semantic sort of S.[4] *Relative stability* is now defined.

Definition 3-9. P is *relatively stable* with respect to Q if P has no τ_Q derivatives.

Now, to prove that LII is preserved by relational composition, it will be necessary to allow only initially stable agents with initial implementations relatively stable.

Definition 3-10. The agent pair (I, S) meets the *conformational stability* (CS) *assumption* if S is stable and I is relatively stable with respect to S.

Lemma 3-10. *Relational composition preserves LII for weak conformations under the CS assumption.*

Proof: Write $P\ \mathcal{V}\ Q\ \mathcal{W}\ R$ for weak conformations $\mathcal{V}$ and $\mathcal{W}$. One must establish $\forall \gamma \in \mathcal{A}(R)$ that whenever $P \overset{\gamma}{\rightarrow} P'$ and $R \overset{\gamma}{\Rightarrow}$ then $R \overset{\gamma}{\Rightarrow} R'$ with $P'\ \mathcal{VW}\ R'$.

- Since Q is relatively stable with respect to R, LSAI requires that $Q \overset{\gamma}{\Rightarrow}$ immediately.
- $\therefore Q \overset{\gamma}{\Rightarrow} Q'$ with $P'\ \mathcal{V}\ Q'$ by LII.
- In turn, $R \overset{\gamma}{\Rightarrow} R'$ with $Q'\ \mathcal{W}\ R'$ by LIAI.

---

[4] The potential ambiguity with τ_s (the synchronization of s and $\bar{s}$) is avoided since the relative τ subscript is an agent name (capital letter) instead of an action label (lower case letter).

- Hence $P' \; \mathcal{VW} \; R'$.

◻

Proposition 3-11. *Relational composition preserves weak conformation under the CS assumption.*

Proof: Propositions 3-7, 3-8, 3-9 and 3-10. ◻

3.5 Summary

This chapter presented the *weak conformations* as a set of properties derived from intuitive notions of compliance. Proposition 3-11 demonstrated that relational composition ∘ preserves weak conformation, but not without cost. The *CS* assumption—more general than Milner's initial stability condition—had to be invoked.

Definition 3-8 is coinductive, referring recursively to $I' \; \mathcal{W} \; S'$ in each law, with no primordial pair offered as a basis. As such, many relations qualify as weak conformations, including the empty relation. The largest, *weak conformance*, is introduced in next chapter.

IV. Weak Conformance and Congruent Weak Conformance

This chapter presents the largest of the weak conformations, called *weak conformance*, or $\succeq_w$, and develops formal results for weak conformance. Progress is begun toward showing weak conformation to be a congruence, but the attempt stalls pending further restrictions to CCS models. Weak conformance is thus refined to *congruent weak conformance* $\succeq_w$ by placing reasonable design restrictions on CCS models. Far from being severe, these restrictions are shown to be quite consistent with good design intent, prohibiting dubious practices. Congruent weak conformance is then proven to be both a partial order and a congruence. Partial orders that are congruent are commonly called *precongruences*. As a precongruence, $\succeq_w$ serves as a correct model of safe substitution.

4.1 Weak Conformance

 Just as $\approx$ is the largest *weak bisimulation*, *weak conformance* is the largest weak conformation.

Definition 4-1. *Weak conformance* $\succeq_w \equiv \cup\{\mathcal{W}: \mathcal{W}$ is a weak conformation$\}$.

Proposition 4-1. $\succeq_w$ *is a weak conformation.*

Proof: Union preserves weak conformation. ❑

Proposition 4-2. $\succeq_w$ *is the largest weak conformation.*

Proof: Any weak conformation $\mathcal{W} \subseteq \succeq_w$ as a result of Definition 4-1. ❑

Proposition 4-3. $\succeq_w$ *is reflexive.*

Proof: As a weak conformation, $Id_p \subseteq \succeq_w$ and hence $P \succeq_w P$ for all P. ❑

Proposition 4-4. *Under the CS assumption,* $\succeq_w$ *is partial order.*[1]

Proof:

- *Reflexivity*. Proposition 4-3.

- *Transitivity*. Given $P \succeq_w Q \succeq_w R$, $P \succeq_w R$ follows immediately since the relational composition $\succeq_w \succeq_w$ is a weak conformation and $\therefore$ $\succeq_w \succeq_w$ $\subseteq$ $\succeq_w$.

- *Antisymmetry*. Given $P \succeq_w Q$ and $Q \succeq_w P$, observe that both P and Q are stable under the CS condition, and that no extraneous actions are possible.

    For inputs and τ, if $P \xrightarrow{a} P'$ then $Q \xRightarrow{a} Q' \succeq_w P$ by LSIT.

    For outputs, if $P \xrightarrow{\bar{a}} P'$ then $Q \xRightarrow{\bar{a}} Q' \succeq_w P$ by LIOT.

    Hence Q simulates P.

    Similarly, P simulates Q and a weak bisimulation exists between the two.

    Thus $P \approx Q$ and, since both are stable, $P=Q$.

    $\blacksquare$

The unmodified weak conformance is not a partial order—the CS condition must be invoked. Nor is the unmodified weak conformance a *congruence*. However, appropriate restrictions to CCS models will repair this deficiency, such that congruence *can* be established. The refined property will be called *congruent weak conformance*.

---

[1]   The CS assumption must be invoked for any proposition that relies on, or inherits a reliance on, the relational composition of weak conformations.

4.2 Weak Conformation up to Weak Conformance

A special type of relation called *weak conformation up to weak conformance* will prove to be a useful proof tool. The intuition behind this concept involves the use of $\succeq_w$ to populate a sparse process relation.

For example, suppose a relation X contains only a single pair (P, Q). If one believes that X expresses some sort of compliance relationship, and there are other processes $R \succeq_w P$, then one may suppose that these processes also share that same compliance notion with Q. One might wish to add the pairs (R, Q) to X. In fact, $(R, Q) \in \succeq_w X$. Furthermore, since $P \succeq_w P$, $(P, Q) \in \succeq_w X$ as well. Hence $\succeq_w X$ has the effect of adding pairs to X where any $R \succeq_w P$ replaces P. Though R is not itself X-compliant to Q, one can say R is X-compliant "within a $\succeq_w$," or "up to $\succeq_w$." Similarly, one can continue to augment the relationship with pairs created by replacing Q with any S such that $Q \succeq_w S$. The resulting relation, $\succeq_w X \succeq_w$, in effect "builds up" X by the transitive closure of $\succeq_w$. If $\succeq_w X \succeq_w$ forms a weak conformation, then X is the seed of that weak conformation, and X is called a *weak conformation up to $\succeq_w$*.

Weak conformations up to $\succeq_w$ are useful because they are contained within $\succeq_w$, and this fact makes an important proof tool. Occasionally, it is easier to show that two processes share a weak conformation up to $\succeq_w$ instead of $\succeq_w$ directly. Nevertheless $\succeq_w$ follows immediately.

Definition 4-2. Relation $\mathcal{W}$ is a *weak conformation up to weak conformance* if $\forall \alpha \in \mathcal{A}(S) \cup \{\tau\}, \forall \beta \in \overline{\mathcal{A}}(I) \cup \{\tau\}, \forall \gamma \in \overline{\mathcal{A}}(S), I \, \mathcal{W} \, S$ implies the following four laws:

LSIT'. Whenever $S \xrightarrow{\alpha} S'$ then $\exists t \in (\mathcal{A}(S) \cup \overline{\mathcal{E}x\!tr}(I,S))^*$ such that

(1) $I \xRightarrow{t} I'$

(2) $t \!\upharpoonright\! \mathcal{A}(S) = \hat{\alpha}$

(3) $I' \succeq_w \mathcal{W} \succeq_w S'$

LSO'. Whenever X is a maxoctset of S, $\exists s \in X$ and $\exists t \in \overline{\mathcal{A}}(I)^+$ such that

(1) $S \overset{s}{\Rightarrow} S'$

(2) $I \overset{t}{\Rightarrow} I'$

(3) $t \upharpoonright \overline{\mathcal{A}}(S) = s$

(4) $I' \succeq_w \mathcal{W} \succeq_w S'$

LII'. Whenever $I \overset{\gamma}{\rightarrow} I'$ then

(1) $S \overset{\gamma}{\Rightarrow} S'$

(2) $I' \succeq_w \mathcal{W} \succeq_w S'$

LIOT'. Whenever $I \overset{\beta}{\rightarrow} I'$ and $\delta \equiv \beta \upharpoonright \overline{\mathcal{A}}(S)$ then

(1) $S \overset{\delta}{\Rightarrow} S'$

(2) $I' \succeq_w \mathcal{W} \succeq_w S'$

These "up to" laws differ from the weak conformation laws only in the relationship of the derivative states—that relationship being '$\succeq_w \mathcal{W} \succeq_w$' instead of '$\mathcal{W}$'. The "primed" designation highlights this similarity, which is exploited to quickly execute proofs.

Proposition 4-5. *All weak conformations are weak conformations up to $\succeq_w$.*

Proof: Each "unprimed" law $N \in \{LSIT, LSO, LII, LIOT\}$ has a conclusion $I' \mathcal{W} S'$. Rewrite it as $I'\ Idp\ \mathcal{W}\ Idp\ S'$. Since $Idp \subseteq \succeq_w$ one derives $I' \succeq_w \mathcal{W} \succeq_w S'$ thus establishing the corresponding law N'. $\qquad\qquad\blacksquare$

Proposition 4-6. *If $\mathcal{W}$ is a weak conformation up to $\succeq_w$ then $\succeq_w \mathcal{W} \succeq_w$ is a weak conformation under the CS assumption.*

Proof: Show that $I \succeq_w \mathcal{W} \succeq_w S$ satisfies the weak conformation laws.   For each "unprimed" law $N \in \{LSIT, LSO, LII, LIOT\}$:

- Write $I \succeq_w P \; \mathcal{W} \; Q \succeq_w S$.

- Apply Law N to each $\succeq_w$ and Law N' to $\mathcal{W}$.

- The resulting derivative relationships are: $I' \succeq_w P'$, $P' \succeq_w \mathcal{W} \succeq_w Q'$, and $Q' \succeq_w S'$.

- By composition $I' \succeq_w \succeq_w \mathcal{W} \succeq_w \succeq_w S'$, which reduces to $I' \succeq_w \mathcal{W} \succeq_w S'$.

- Hence $\succeq_w \mathcal{W} \succeq_w$ satisfies Law N.

$\square$

Proposition 4-7. *If $\mathcal{W}$ is a weak conformation up to $\succeq_w$ then $\mathcal{W} \subseteq \; \succeq_w$ under the CS assumption.*

Proof: $\mathcal{W} \; = \; Id_p \; \mathcal{W} \; Id_p \;\; \subseteq \;\;\; \succeq_w \mathcal{W} \succeq_w \;\; \subseteq \;\; \succeq_w$ $\square$

Proposition 4-7 is the result that will serve as a useful proof tool. To show that $I \succeq_w S$ it suffices to show that a weak conformation up to $\succeq_w$ exists between I and S.

Proposition 4-8.   *All bisimulations (including $\approx$ and $\sim$) are weak conformations (and by Proposition 4-5, they are weak conformations up to $\succeq_w$ also).*

Proof: There are no extraneous actions between bisimilar processes; and the back and forth laws of bisimulation are stricter than the weak conformation laws. Thus the proof of each weak conformation law is straightforward. $\square$

4.3 Congruent Weak Conformance

Milner found that $\approx$ is not a congruence over the unmodified CCS, so he constructed the slightly finer $=$ to serve as a congruence.   Similarly, $\succeq_w$ is not a

congruence over CCS, and a slightly finer conformation is needed. Thus a *congruent weak conformance* (to be symbolized as '$\succeq_w$') is desired.

One restriction has already been imposed—the CS assumption—to assure the relational composition of weak conformations. The CS assumption disposes of initial instability, a difficulty faced by $\approx$ as well.

An additional difficulty stems from the possibility of extraneous actions being "promoted" to specified actions during the construction of compound agents. This is not an issue for equivalences, where there are no extraneous actions. To achieve a congruent weak conformance over CCS constructions, one must prevent extraneous actions from being promoted during the course of the construction. Extraneous actions *prior to* the construction must remain extraneous *after* the construction, unless they disappear entirely from the sort of the composite implementation. Suppose one has $S \stackrel{\text{def}}{=} \bar{b}.NIL$ and $I \stackrel{\text{def}}{=} \bar{b}.\bar{c}.NIL + d.NIL$. A weak conformation exists between specification S and implementation I. The CS assumption holds, and $\bar{c}$ and d are allowable extraneous actions. Clearly, however, $\bar{c}.I$ does *not* conform to $\bar{c}.S$ nor does $d.I$ conform to $d.S$. The Prefix operation has "promoted" extraneous actions $\bar{c}$ and d[RWB1].

All the CCS constructors (except Restriction) can create problems by unwittingly promoting extraneous actions to specified actions. Therefore, to obtain a congruent weak conformance relation, CCS constructions need to be constrained to disallow such promotion. This prohibition is not an unworkable limitations. Rather, it is consistent with good design sense. Indeed, the behavior of an extraneous pin is, by its very nature, a "don't care" issue. To suddenly levy requirements on the "don't care" pin at a higher level of abstraction represents a questionable change in designer intent.

Definition 4-3. Let $(\widetilde{I}, \widetilde{S})$ be an indexed system of agents such that $I_i \mathcal{W}_i S_i$ for weak conformation $\mathcal{W}_i$. Let $E\{\widetilde{X}\}$ be a CCS expression on multiple agents denoted by indexed

variable $\widetilde{X}$. $E\{\widetilde{X}\}$ can employ *all* constructors *except* Restriction and Relabeling. $E\{\widetilde{X}\}$ meets the *Preservation of Extraneous Action* (*PEA*) condition if:

(1) $\mathcal{E}xtr(I_i, S_i) \subseteq \mathcal{E}xtr(E\{\widetilde{I}\}, E\{\widetilde{S}\})$ for all indices i.

(2) $\overline{\mathcal{E}xtr}(I_i, S_i) \subseteq \overline{\mathcal{E}xtr}(E\{\widetilde{I}\}, E\{\widetilde{S}\})$ for all i.

PEA guarantees that all extraneous actions remain extraneous after the construction $E\{\widetilde{X}\}$ so that unreachable paths are not inadvertently activated.

In addition to PEA, one needs assurance that no *coactions* are introduced that can synchronize with extraneous actions. Such synchronization can activate unreachable paths via the silent action τ. This necessitates yet another design constraint:

Definition 4-4. Under the same assumptions as Definition 4-3, if, for all Parallel Compositions $(E_1\{\widetilde{X}\}|E_2\{\widetilde{X}\})$ within $E\{\widetilde{X}\}$:

(1) $\forall a \in \mathcal{A}(E_1\{\widetilde{S}\}), \bar{a} \notin \overline{\mathcal{E}xtr}(E_2\{\widetilde{I}\}, E_2\{\widetilde{S}\})$

(2) $\forall \bar{a} \in \overline{\mathcal{A}}(E_1\{\widetilde{S}\}), a \notin \mathcal{E}xtr(E_2\{\widetilde{I}\}, E_2\{\widetilde{S}\})$

(3) $\forall a \in \mathcal{A}(E_2\{\widetilde{S}\}), \bar{\alpha} \notin \overline{\mathcal{E}xtr}(E_1\{\widetilde{I}\}, E_1\{\widetilde{S}\})$

(4) $\forall \bar{a} \in \overline{\mathcal{A}}(E_2\{\widetilde{S}\}), a \notin \mathcal{E}xtr(E_1\{\widetilde{I}\}, E_1\{\widetilde{S}\})$

then $E\{\widetilde{X}\}$ meets the *Extraneous Synchronization Prohibition* (*ESP*) with respect to $(\widetilde{I}, \widetilde{S})$.

Given the design constraints (CS, PEA and ESP) introduced thus far, one now proceeds to prove that each combinator preserves $\succeq_w$, given the constraints. The propositions are stated as generally as possible, invoking only the necessary condition(s), and applying to general weak conformations when possible, and to $\succeq_w$ alone only when necessary.

Proposition 4-9.   *A weak conformation $\mathcal{W}$ is preserved by the Prefix combinator under the PEA restriction.*

Proof: Given $I \ \mathcal{W} \ S$, show that $\forall \alpha \in \mathcal{A}ct : \alpha.I \ \mathcal{W} \ \alpha.S$.   Observe that α is the only immediate action that $\alpha.I$ and $\alpha.S$ can perform, and PEA assures that $\alpha \notin (\overline{\mathcal{E}xtr}(I,S) \cup \mathcal{E}xtr(I,S))$. Thus α lies within *both* $\mathcal{L}(I)$ and $\mathcal{L}(S)$, or *neither*.

- *LSIT.* $\alpha \in \mathcal{A}(S) \cup \{\tau\}$. $\alpha.S \overset{\alpha}{\rightarrow} S$, $\alpha.I \overset{t}{\Rightarrow} I$ using $t = \hat{\alpha}$. The target states are $I \ \mathcal{W} S$.

- *LII and LIOT.* Similar.

- *LSO.* $\alpha \in \overline{\mathcal{A}}(S)$. Let X be a maxoctset of S.

    $Y \equiv \{\alpha.s : s \in X\}$ is a maxoctset of $\alpha.S$.

    By LSO, $\exists r \in \overline{\mathcal{A}}(I)^+$ such that r implements some $x \in X$ where

    $$I \overset{\varepsilon}{\Rightarrow} I', \ \ S \overset{\varepsilon}{\Rightarrow} S', \ \ r \restriction \overline{\mathcal{A}}(S) = x, \ \ I' \ \mathcal{W} S'.$$

    Now $\alpha.I \overset{\alpha}{\rightarrow} \overset{\varepsilon}{\Rightarrow} I'$ and $\alpha.S \overset{\alpha}{\rightarrow} \overset{\varepsilon}{\Rightarrow} S'$ with $\alpha.x \in Y$.

    If $\alpha \notin r$ then $r \restriction \overline{\mathcal{A}}(\alpha.S) = r \restriction \overline{\mathcal{A}}(S) = x$.

    If $\alpha \in r$ then by PEO $a \notin \overline{\mathcal{E}xtr}(I,S)$ and $\therefore \alpha \in \overline{\mathcal{A}}(S) = \overline{\mathcal{A}}(\alpha.S)$. Again,

    $$r \restriction \overline{\mathcal{A}}(S) = x.$$

    Since $r \restriction \overline{\mathcal{A}}(S) = x$ independent of whether $\alpha \in r$ then $\alpha.r \restriction \overline{\mathcal{A}}(\alpha.S) = \alpha.x$.

    Now $\alpha.x \in Y$ and $\therefore \ \alpha.r$ is an implementation of $\alpha.x \in Y$.

    ∎

To show that Summation or Choice preserves a weak conformation, one must know how the maxoctsets of the Summation are formed from the maxoctsets of the components. Hence the next lemma:

Lemma 4-10.   *Let M be a maxoctset of $R = S + T$.   Let $M = M_S \cup M_T$ where $M_S = \{s \in M : S \overset{\varepsilon}{\Rightarrow} \}$ and $M_T = \{s \in M : T \overset{\varepsilon}{\Rightarrow} \}$. M_S and M_T are maxoctsets of S and T, respectively.*

Proof: By contradiction.

- $\forall x, y \in M : R \overset{x}{\Rightarrow} \approx R'$, $R \overset{y}{\Rightarrow} \approx R'$ and $x =_{conf} y$ since M is a maxoctset.

- $\forall x, y \in M_S : R \overset{x}{\Rightarrow} \approx R'$, $R \overset{y}{\Rightarrow} \approx R'$ and $x =_{conf} y$ since $M_S \subseteq M$.

- Since x, y belong to S, $S \overset{x}{\Rightarrow} \approx R'$, $S \overset{y}{\Rightarrow} \approx R'$

- Thus M_S is at least an *octset* of S. Similarly, M_T is an octset of T.

- *Trial Hypothesis.* Assume one is not a *maxoctset.* W.l.o.g. let it be M_S.

- Then S must have some maxoctset M_S' with respect to some $s.s'$ where $s' \neq \varepsilon$.

- $\forall x' \in M_S'$: let $x' = y.z$, where $y \in M_S$ and $z =_{conf} s'$.

- Let $S \overset{y}{\Rightarrow} S' \overset{z}{\Rightarrow} S''$.

- $\forall x \in M : S + T \overset{x}{\Rightarrow} \approx R'$.

- Since $y \in M$, one must have $R' \approx S'$.

- Since $S' \overset{z}{\Rightarrow} S''$ then $R' \overset{z}{\Rightarrow} R'' \approx S''$.

- $\therefore R = S + T$ has an octset with respect to $x.z$ and M cannot be a maxoctset of R.

$\Rightarrow\Leftarrow$

Proposition 4-11. *A weak conformation $\mathcal{W}$ is preserved by Summation under the PEA assumption.*

Proof: Given $I \ \mathcal{W} \ S$ and $J \ \mathcal{W} \ T$, show that $I + J \ \mathcal{W} \ S + T$, assuming w.l.o.g. that any transition out of $S + T$ has S as its source.

- LSIT. $S + T \overset{\alpha}{\Rightarrow} S'$ for $\alpha \in \mathcal{A} \cup \{\tau\}$.

  By LSIT, $I \overset{\alpha}{\Rightarrow} I' \ \mathcal{W} \ S'$. Yet if $I \overset{\alpha}{\Rightarrow} I'$ then $I + J \overset{\alpha}{\Rightarrow} I'$.

  To show that $t \upharpoonright \mathcal{A}(S + T) = \hat{\alpha}$, assume otherwise:

  $\exists \bar{a} \in t$ such that $\bar{a} \in \overline{\mathcal{A}}(T)$ while $\bar{a} \notin \overline{\mathcal{A}}(S)$.

  This violates PEA.

  $\Rightarrow\Leftarrow$

- *LII* and *LIOT*. Similar.

- *LSO*. Apply Lemma 4-10. Any maxoctset of $S + T$ is of the form $M = M_S \cup M_T$.

  M_S and M_T are maxoctsets of S and T, respectively.

  M_S and M_T each must contain at least one implemented string.

  $\therefore M$ must contain at least *two* implemented strings.

  If $s \in M_S$ is implemented by I, then $I \overset{\cdot}{\Rightarrow} I' \, \mathcal{W} \, S'$.

  $\therefore I + J \overset{\cdot}{\Rightarrow} I'$ for $S + T \overset{\cdot}{\Rightarrow} S'$ thus $I + J$ implements s.

  The argument for $s \in M_T$ is similar.

◼

To show that Parallel Composition preserves weak conformation, one must know how maxoctsets of a Parallel Composition are formed from the maxoctsets of the components. Hence Lemma 4-12 is given to aid Proposition 4-13.

Lemma 4-12. *Let $Y_1...Y_n$ be all the maxoctsets of S and let $Z_1...Z_m$ be the all the maxoctsets of T. Let $s_1...s_n$ be defining sequences for $Y_1...Y_n$, respectively. Similarly, let $t_1...t_m$ be defining sequences for $Z_1...Z_m$. The maxoctsets of $(S \mid T)$ are precisely the nm octsets with respect to $s_i.t_j$, for $1 \le i \le n$ and $1 \le j \le m$.*

Proof: See Appendix B.

Proposition 4-13. *A weak conformation $\mathcal{W}$ is preserved by Parallel Composition under the PEA and ESP conditions.*

Proof: See Appendix B.

Next, consider Restriction and, in particular, Restriction of inputs. One can safely Restrict *specified inputs*, since their removal does not affect the ability of the

implementation to obey LSIT and LII with respect to the remaining inputs. *Extraneous* inputs can also be safely Restricted. Naturally, one can also Restrict any input symbols *external* to *both* the specification and implementation, since such Restriction does not modify the base agents at all. In conclusion: there is *no* limitation on Restricting *inputs*.

Now consider the Restriction of *outputs*. First note that *specified* outputs can be safely Restricted. In the case of the LSO law, Restricting a single specified output action will remove from consideration every maxoctset in which it participates, since that action must appear in every string of that maxoctset. The Restriction of outputs *external* to *both* sorts is moot, as was the case for inputs. However, one must take care when Restricting *extraneous* outputs, for they can appear within implementing strings, and their Restriction will block these strings. The only extraneous output actions that can be safely be Restricted are those whose *only* occurrences lie along unreachable paths. This special type of extraneous output is called an *idle* output action:

Definition 4-5. $\overline{a} \in \overline{\mathcal{Extr}}\,(I,S)$ is an *idle output action* of $I \, \mathcal{W} \, S$ if for every derivative I' of I, whenever $I' \xrightarrow{\overline{a}}$, $\nexists S'$ a derivative of S such that $I' \, \mathcal{W} \, S'$.

Definition 4-6. $\overline{\mathcal{Idle}}\,(I \, \mathcal{W} S) \equiv \{\,\overline{a}\; :\; \overline{a}$ is an idle output action of $I \, \mathcal{W} S.\,\}$.

One can now define the conditions required to achieve congruence for weak conformations under Restriction.

Definition 4-7. The label set $L \in \mathcal{L}$ meets the *Congruent Output Restriction* (*COR*) condition with respect to a weak conformation $I \, \mathcal{W} \, S$ if $\forall a \in L$, $a,\overline{a} \notin \overline{\mathcal{Extr}}\,(I,S) - \overline{\mathcal{Idle}}\,(I \, \mathcal{W} S)$.

The COR condition forbids the Restriction of extraneous outputs, unless they are idle. Thus three kinds of outputs may be safely restricted: (1) *specified* outputs, (2) *idle* outputs and (3) outputs *external* to both $\overline{\mathcal{A}}(I)$ and $\overline{\mathcal{A}}(S)$, whose restriction is moot.

Proposition 4-14. *Restriction preserves a weak conformation when the COR condition is met.*

Proof: See Appendix B.

Finally, consider *Relabeling*. This operator can cause congruence failure if the Relabeling function assigns the same name to previously distinct symbols. Therefore, one must require that the Relabeling function be one-to-one, in other words, an *injection.* Also, those signals not explicitly renamed are implicitly renamed to their "old" names, which of course must not collide with any "new" names, so the function must in fact be a *bijection.*

Definition 4-8. A Relabeling function meets the *bijective relabeling* (*BR*) condition if it is a bijection.

Proposition 4-15. *Relabeling preserves weak conformation under the BR condition.*

Proof: Similar to Proposition 4-14.

It remains to show that congruent weak conformance is preserved under recursive definition. This cannot be shown for general weak conformations, or even for $\succeq_w$. It *can* however be demonstrated for $\succeq_w$ when the CS, PEA, ESP, COR and BR conditions apply.

Proposition 4-16. *If* $\tilde{A} \stackrel{def}{=} \tilde{P}$ *then* $\tilde{A} \succeq_w \tilde{P}$.

Proof: $\widetilde{A} \stackrel{def}{=} \widetilde{P}$ implies $\widetilde{A} \sim \widetilde{P}$ (Milner, 1989:Propostion 12.11). Since $\sim$ is a weak conformation then $\sim \subseteq \succeq_w$. $\widetilde{A} \succeq_w \widetilde{P}$ follows. $\qquad\qquad\qquad$ ◻

Proposition 4-16 allows one to conduct the proof of Proposition 4-17 with respect to a single variable. It is understood that Proposition 4-17 will be easily extended to the multivariate case. The proof of Proposition 4-17 is by transition induction (Milner, 1989:58, 100), a form of coinduction (Wegner and Goldin, 1999).

Proposition 4-17. *Let $E \succeq_w F$ where expressions E and F contain at most the single variable X. Under the CS, PEA, ESP, COR and BR conditions, whenever $I \stackrel{def}{=} E\{I/X\}$ and $S \stackrel{def}{=} F\{S/X\}$ then $I \succeq_w S$.*

Proof: See Appendix B.

Though $\succeq_w$ itself is not a congruence, the previous propositions show that $\succeq_w$ *is* preserved by each combinator, *if* the five design restrictions apply. Hence, a weak conformation exits, slightly finer than $\succeq_w$, which *is* a congruence. This desired relation is called *congruent weak conformance* $\succeq_w$.

Definition 4-8. Let $I \succeq_w S$. Furthermore assume that the CS, PEA, ESP, COR and BR conditions apply. Then *I* and *S* enjoy the *congruent weak conformance* relation, written $I \succeq_w S$.

Proposition 4-18. $\succeq_w$ *is a congruence.*

Proof: Propositions 4-9, 4-11,4-13, 4-14, 4-15, 4-17. $\qquad\qquad\qquad$ ◻

Proposition 4-18 is a major result, establishing $\succeq_w$ as a correct model for safe substitution. It now remains to justify the earlier conjecture that $\succeq_w$ is a partial order. Now $\succeq_w$ is already known to be a partial order under the CS assumption. As a refinement of weak conformance that includes CS among other conditions, it follows that $\succeq_w$ is a partial order.

Proposition 4-19. $\succeq_w$ *is a partial order.*

Proof: Proposition 4-4.   ◻

4.4 Summary

In the last chapter a set of properties called *weak conformations* were established under four laws that embody intuitive notions of hardware compliance. In this chapter, the largest of the *weak conformations* was designated *weak conformance* and given the symbol $\succeq_w$. To serve as a model for safe substitution of hardware, $\succeq_w$ had to be shown to be a *congruence*, *i.e.*, that it be preserved by all CCS operators. To achieve this goal, it was necessary to limit CCS constructions by the five conditions: CS, PEA, ESP, COR and BR. Happily, these conditions are consistent good design intent. Weak conformance, when refined by these five conditions, yields the property of *congruent weak conformance* $\succeq_w$.

Congruent weak conformance $\succeq_w$ was developed and proven to be a congruent partial order, or *precongruence*. This precongruence derives maximal flexibility and embodies all weaknesses in input, output, and no-connect signals via the four transitional laws of weak conformation. Five construction restrictions assure that $\succeq_w$ is a fully replaceable model of conformance to specification. This is the best formal relation known for verifying implementations against specifications.

In the next chapter, a hypothetical VHDL-to-CCS translator is validated using congruent weak conformance.

V. VHDL-to-CCS Translation

5.1 Introduction

In the previous chapter, the property of congruent weak conformance $\succeq_w$ was proven to be a precongruence, and therefore a correct model of safe substitution. The present chapter applies $\succeq_w$ to a practical problem: the translation of VHDL code to CCS. The two languages have different semantics, so the translation problem is challenging.

VHDL has informal *simulation semantics*, which are defined in the *VHDL Language Reference Manual* (IEEE, 1992). The semantics of CCS are given by its transition rules (Milner, 1989: 45, 57). Fundamental differences between VHDL and CCS semantics are: (1) *simulation* versus *bisimulation* semantics, (2) *quantitative* versus *indefinite* time, (3) complete *hiding* of internal action versus *abstraction* of hidden action to τ, (4) *broadcast* versus *handshake* communication, (5) *level-signal* versus *transitional* semantics and (6) *simultaneity* versus *interleaving* concurrency.

5.1.1 Simulation and Bisimulation Semantics. The VHDL simulation cycle is a three part repeating sequence that (1) responds to current events, (2) posts future events as *transactions*[1] onto the *drivers* (event lists) that correspond to each signal, and then (3) advances the simulation clock to the next scheduled transaction (Lipsett and others, 1989: 12-13). The *Language Reference Manual* defines the meaning of each VHDL construct by how an event-based simulator interprets and executes it. Since the simulation cycle is so central to VHDL semantics, formal models of VHDL code often include a formal model of a simulation engine (Fujita and others, 1983; 1983a; Read and Edwards, 1994; van Tassel, 1994).

CCS, on the other hand, supports *bisimulation semantics* in which processes are differentiated by the actions they can potentially perform.

---

[1] An *event* is a signal transition that has actually been accomplished by the simulator. *Transactions*, on the other hand, represent potential future events. The simulator removes transactions from the signal drivers and creates events.

5.1.2 Time. For VHDL, events are separated in quantitative time by a simulation clock, and the duration between events can be precisely calculated. An infinitesimal quantity of time called *delta* is also supported (Bhasker, 1999: 74). Delta is tied to the simulation cycle, and corresponds to the minimum advance of the simulation clock. When a transaction is first posted to a signal driver, and no delay is specified, the simulation cycle must nevertheless complete a repetition in order to react to it and produce an event. Each advance of one delta corresponds to one repetition of the simulation cycle when the simulation clock itself does not advance. Thus, events can be separated by one or more deltas while occurring at the same simulation time. Deltas enforce a strict ordering among otherwise simultaneous events or transactions. A transaction scheduled two deltas after the present is considered to be strictly *later* than one scheduled one delta later. However, no number of deltas can exceed the smallest *finite* delay time.

Although CCS can express the *ordering* of actions, *duration* between actions is indeterminate since time is not quantified. Pending actions simply occur at some indefinite future time.

5.1.3 Abstraction. When hierarchical models are built in VHDL, internal signals between components disappear entirely from the port list of the composite. Thenceforth, such internal actions can neither be manipulated nor observed by the environment.

For CCS, internal action, though hidden, remains expressed at the top level as τ. A τ cannot be manipulated or *directly* observed, but it does have an effect on observed behavior due to its ability to preempt potential actions.

5.1.4 Communication. VHDL allows *broadcast* communications. Several wires can connect to a single node. Thus, multiple processes or components can read the value of a single signal. That signal need not be explicitly split to service each process or component individually. The ability of a single output signal to drive multiple processes

is commonly is called *fanout*. Similarly, two or more processes can drive a single node in VHDL if the designer provides a *resolution function* (Bhasker, 1999: 111).

CCS, on the other hand, uses *handshake* communication. Such communication is strictly one-to-one and occurs when one agent offers and action and another the corresponding coaction. If two receivers attempt to handshake on the same action, one of them fails to communicate. Thus, a lone signal cannot directly drive multiple devices. Multiple fanout must be modeled indirectly by providing a FORK agent to explicitly provide multiple copies of an action to communicate with all offered coactions. Yet the FORK still fails to accurately model the broadcast communication of VHDL. For VHDL, a single event along a multiply connected node is simultaneous along all branches. For the CCS FORK, occurrences along each branch must be ordered, and the time separation between them is undetermined.

By requiring FORKs, pure CCS supports the *delay-insensitive* hazard model of asynchronous design (Seitz, 1980: 246). The many branches of a wire node are modeled as operating independently. VHDL communication, on the other hand, matches the *speed-independent* hazard model, where the delays along the various branches of a node are presumed so close as to be negligible (Seitz 1980: 250). A single signal passes through the various branches at essentially the same time.

A modified version of CCS adds operators to achieve the ability to model broadcast communication (Stevens, 1994: 180-5). In particular, this version adds a *conjunction* operator $|_c$ to model broadcast communication among parallel agents. The conjunction operator is similar to the $||$ operator of CSP (Hoare, 1985).

The delay-insensitive model is very strict. The designs that can be produced under its regimen are very few. Thus, the broadcast version of CCS thus makes a more practical translation target. In this chapter, the consequences of translation to both versions of CCS are explored.

5.1.5 Level Signals and Transitional Semantics. VHDL assigns explicit level values to its signals and does not simply direct them to "change." Most assignment statements are explicit in this, such as "X <= 1".

In contrast, CCS models *transitions* without specifying what the level values are. Thus when $FIFO \overset{def}{=} in.\overline{out}.FIFO$ one can surmise that *in* and $\overline{out}$ start at '0' and after one iteration they transition to '1', back to '0' after two iterations, and so forth. An equally valid interpretation starts them at opposite values '0' and '1', and they forever transition in opposite directions. In CCS, level values are simply undefined since they are irrelevant to the transitional semantics.

Now VHDL code can take on the *appearance* of transitional instead of level-signal semantics. Assignment statements such as "X <= **not** X" are quite legal. Nevertheless the resulting transaction must be to a level value. Thus, the present value of X must be noted, and an assignment to the opposite value posted to the driver. One cannot merely schedule X to "switch value." Even though VHDL code can appear transitional, the underlying simulation semantics does not support it.

Interestingly enough, a VHDL transaction can be ineffective in creating a real transition. For example, if a transaction is scheduled to drive X to '1', yet, due to previous events, X is already at '1', then no real transition occurs.

5.1.6 Simultaneity versus Concurrency. VHDL allows simultaneous events, since transactions can bear the same time stamp. Microscopic ordering by delta delays is not even required. Transactions can indeed be scheduled to occur on the same delta cycle.

CCS however, engages in *interleaving concurrency*. CCS does not recognize absolute simultaneity, taking the view that a fine enough division of time will unveil an ordering among seemingly simultaneous actions. CCS actions *can*, however, be considered *concurrent*. Concurrent actions are not necessarily simultaneous. Rather, their relative order is simply indeterminate. Therefore, one writes *x.y* + *y.x* or, more compactly, (*x*|*y*) to indicate this uncertainty.

5.2 Translation Rationale

Due to semantic mismatches such as those given in section 5.1, translation from one semantic system to another, by its very nature, cannot preserve all meaning. Some information must necessarily be lost, whether it be explicit simulation time, the ability to model simultaneity, the ability to discriminate based on bisimulation, etc.

If information is lost, then why perform such a translation at all? One reason is this: the designer may wish to reuse off-the-shelf component models, and they may not all be available in a single language. Secondly, the designer may wish to exploit the verifications that another system can offer. These verifications may be more accurate, more efficient, or use a stricter semantics than is possible in the source language.

Yet if transformed models are not semantically equivalent to their originals, how can post-transformational verifications be deemed valid? Such verifications are valid if the translation process preserves an appropriate property. Since the ultimate goal of a designer is to "substitute" his design for a specification, then *safe substitution* (or *congruence*) is the property that must be preserved by the translation. As the loosest known congruence modeling device compliance, $\succeq_w$ is thus the property that must be preserved by inter-semantic translation.

This chapter shows how $\succeq_w$ can be used to validate such transformations. First, some very basic agents are identified. The agents are detailed enough to represent the salient features of the CCS and VHDL semantics, and will be used to illustrate the translation process. Both VHDL and CCS models will be proposed for these agents. By comparing these models, the characteristics of a VHDL-to-CCS translator will be derived. The translation rules discovered during the course of this exercise will then be enumerated. These transformations will then be shown to preserve $\succeq_w$.

The VHDL-to-CCS translator discussed in this chapter is not an implemented translator. The purpose of this chapter is to infer the general characteristics of such a

translator. This serves two purposes: (1) to demonstrate the *feasibility* of such a translator and (2) to show that such a translator with these characteristics preserves $\succeq_w$.

5.3 Translation Models

Consider three buffer specifications *F*, *G*, and *H*. Each is a one-token buffer. *F* is the same as the familiar one-place FIFO (Equation 2-3). *G* and *H* are contrived examples created to provide instances of extraneous input and output. *G* is like *F* except that *G* produces duplicate outputs concurrently, rather than a single output. Hence *G* has a maxoctset $\{\overline{o}.\overline{p}, \overline{p}.\overline{o}\}$. *H* has two input lines as well as two output lines. One input requests *F*-like behavior, *i.e.*, one output. The other input requests *G*'s behavior, *i.e.*, two outputs. Unlike *G* however, *H* produces a specific interleaving of the dual outputs, not both alternatives. Thus *H* is an implementation of *G* that takes a single path through the maxoctset in accordance with LSO.

The models for these buffers appear in this section in three distinct groups: (1) *reference* models, (2) *initial* models and (3) *target* models.

The *reference* models appear first. These models merely document the behavior under discussion. For brevity, the reference models are given in CCS. However they are not the subject of translation, since the intent is to study VHDL-to-CCS translation, and not the reverse.

The *initial* models then are VHDL models *inspired* by, but not necessarily faithful to, the reference models. Semantic differences will necessarily limit the fidelity of the initial models to the reference models.

The *target* models are CCS models derived from the initial models by the application of translation rules. Again, semantic differences will limit the fidelity of the target models to the initial models. However, whereas the initial models derive from the reference models by *inspiration*, the target models derive from the initial models by a

disciplined approach. One should not expect the translation process to recover the reference models.

Here are the basic reference models for agents *F*, *G* and *H*:

$$F \stackrel{\mathrm{def}}{=} i.\overline{o}.F \tag{5-1}$$

$$G \stackrel{\mathrm{def}}{=} i.(\overline{o}\,|\,\overline{p}).G \tag{5-2}$$

$$H \stackrel{\mathrm{def}}{=} i.\overline{o}.\overline{p}.H + j.\overline{o}.H \tag{5-3}$$

One can also make two-token buffers using the same protocol. Here are two-token reference models *FF* and *GG* that are analogous to *F* and *G* respectively:

$$FF \stackrel{\mathrm{def}}{=} i.FF1$$
$$FF1 \stackrel{\mathrm{def}}{=} \overline{o}.FF + i.\overline{o}.FF1 \tag{5-4}$$

$$GG \stackrel{\mathrm{def}}{=} i.GG1$$
$$GG1 \stackrel{\mathrm{def}}{=} j.(\overline{o}\,|\,\overline{p}).GG + i.(\overline{o}\,|\,\overline{p}).GG1 \tag{5-5}$$

A two-token buffer *HH* corresponding to *H* could also be defined. Such a buffer would need to differentiate *i* and *j* and remember their arrival sequence to generate $\overline{o}$ and $\overline{o}.\overline{p}$ appropriately. That model is relatively complex and does not contribute to the present discussion.

One can also build the two-token buffers structurally from the one-token buffers using Parallel Composition. Call these composites *FPF, GPG*.[2]

---

[2] Think of '*P*' as representing "parallel."

$$FPF \stackrel{def}{=} (F[m/o]\|F[m/i])\backslash m \qquad\qquad (5\text{-}6)$$

$$GPG \stackrel{def}{=} (G[x/o,m/p]\|G[m/i])\backslash m \qquad\qquad (5\text{-}7)$$

Note that the seven reference models F, G, H, GG, FF, FPF and GPG can be assembled into a structure that is ordered upon the congruent weak conformance relationship, where the arrows point from implementation to specification. First of all, one has $G \succeq_w F$ because G's output $\bar{p}$ is extraneous to F. For the same reason, $GG \succeq_w FF$. Now $H \succeq_w G$ because input j is extraneous to G and because the lone sequence $\bar{o}.\bar{p}$ is a sufficient implementation the maxoctset of G's maxoctset $\{\bar{o}.\bar{p}$, $\bar{p}.\bar{o}\}$. Furthermore, $FF \succeq_w F$ and $GG \succeq_w G$ because the additional states of FF and GG that accept a second token are unreachable by F and G. Also $GPG \succeq_w GG$. Two of the models are in fact observationally congruent: $FPF = FF$. For them, the $\succeq_w$ relationship is bi-directional.

By the transitivity of $\succeq_w$ one can follow the arrows and infer all pairs that share the $\succeq_w$ relationship. Thus *all* the agents are seen to conform to F and thus F is a greatest lower bound or *least specification*. However the diagram is not a lattice and there is no *greatest implementation*. In the absence of H, however, GPG would fulfill that role.

$$GPG \;\rightarrow\; GG \;\rightarrow\; FF \;=\; FPF$$
$$\downarrow \qquad\quad \downarrow$$
$$H \;\rightarrow\; G \;\rightarrow\; F$$
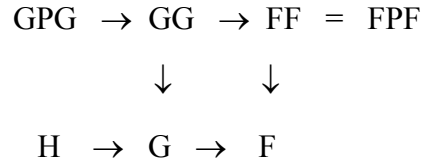
Figure 5-1. Conformance Structure for the Reference Models

These seven reference models now inspire the *initial* models. The initial models are VHDL state machines. One starts with the entity declaration for F:

```
entity F is
    ( I : in bit; O : out bit )
end F;
```

For simplicity, all signals are of type **bit**. These signals will become lower case action labels after translation, with those of mode **out** receiving an overbar or leading apostrophe. However, an unaccompanied entity declaration is incomplete and gives only the *sort* of an agent. One needs an entity-architecture-configuration triple to extract a full agent definition. Such a triple is called a VHDL *design unit*.

```
-- Initial Model for One-token Buffer F

entity F is
    ( I : in bit; O : out bit )
end F;

architecture BEHAVIOR of F is
begin
   process(I)
   begin
      O <= not O;
   end process;
end BEHAVIOR;

configuration CFG_F of F is
     for BEHAVIOR
     end for;
end CFG_F;
```

The **configuration** body tells the VHDL analyzer what design units to use for the subcomponents of an architecture. However, purely *behavioral* models such as this one have no subcomponents, and their **configuration** is trivial. In such cases the designer is not required to supply a **configuration**—the VHDL analyzer will create a default configuration. Default configurations will be assumed for the remaining behavioral models.

Like most CCS agents, VHDL processes are reactive. The behavior in **process** (I) above accepts an input I, generates an output O, then evolves to repeat this cycle forever.

Since the output transition occurs strictly later (by one delta) than the input, the behavior of *F* readily translates to a CCS target model:

$$F \overset{def}{=} i.\overline{o}.F \qquad\qquad (5\text{-}8)$$

A pattern can be discerned already. A signal on a process sensitivity list, which must be an input, translates into the "leadoff" action of a sequential CCS term. Assignments within the body of a process create output actions that are appended to that term. The **end process** statement then denotes the point at which the behavior evolves back to the root agent.

Now consider the initial model for *G*:

```
-- Initial Model for One-token Buffer G

entity G is
    ( I : in bit; O,P : out bit )
end G;

architecture BEHAVIOR of G is
begin
   process(I)
   begin
       O <= not O;
       P <= not P;
   end process;
end BEHAVIOR;
```

The transactions on O and P are scheduled at the same simulation time, one delta after the present. In fact, they are simultaneous. Simultaneity cannot be expressed in CCS so concurrency will have to suffice. Thus, the process unfolds into the target model

$$G \stackrel{\mathrm{def}}{=} i.(\; \bar{o} \mid \bar{p} \;).G \qquad\qquad (5\text{-}9)$$

The translator must detect when VHDL transactions are scheduled at the same time, and translate them as a concurrent ($\bar{o} \mid \bar{p}$) and not a sequential $\bar{o}.\bar{p}$.

The buffer *G* forms an excellent example of how the differing semantics result in imperfect translation. The *concurrency* of the CCS model says only that the order of actions is indeterminate. The CCS model is thus looser than the VHDL model, which specifies that the events are simultaneous.

Here is the initial model for *H*.

```
-- Initial Model for One-token Buffer H


entity H is
   ( I,J: in bit; O,P : out bit )
end H;

architecture BEHAVIOR of H is
begin
   process(I,J)
      begin
         if event'I then
            O <= not O;
            P <= not P after delta;
         else if event'J then
            O <= not O;
         end if;
   end process;
end BEHAVIOR;
```

Here the assignment to P is delayed an additional delta to insure it occurs strictly later than the assignment to O. Thus the O transaction is scheduled one delta after the present, and the P transaction two deltas later.

The modeling of the inputs of *H* exhibits another difficulty in matching the semantics of CCS and VHDL. In the CCS reference model the inputs block each other.

The first input to arrive forces a commitment to one branch of behavior, and the other input cannot be received. If the two inputs arrive simultaneously then one input is arbitrarily accepted and the other blocked. In VHDL, when events occur on both I and J at the same time, both can be accepted, and each can then trigger output transactions. In fact, *one* course of action can be denoted for I, a *second* course of action for J, and a *third* course of action (not simply a combination of the first two) can be denoted when I and J arrive together. However, the idea that I and J arriving together can trigger behavior completely different is not natural for hardware, especially for asynchronous hardware. Though unlikely to occur, such behavior is nonetheless expressible in VHDL, though not in CCS. A CCS tool may be able to detect a blocked input and raise an exception, but the language itself does not allow simultaneous activation of both branches. Thus, here is a rare feature that cannot be translated. In the initial (VHDL) code for *H*, the designer has avoided such difficulty by favoring J over I, allowing only one branch of behavior even if both inputs arrive. This still departs from the intent of the CCS reference model, where *i* is equally likely to be favored over *j*.

The failure to capture in VHDL all the nuances of the CCS reference models is not surprising, given the semantic differences. For the present study however, translation in this direction is not an issue. The question is this: given a *VHDL* model, can a *CCS* model be constructed that preserves enough information such that meaningful verifications can be performed? The focus now is how to extract CCS behavior from the initial VHDL models given thus far.

The ability to extract state machine behavior from a VHDL process is critical to the translation. Once a process is recast as a state machine, it can be directly mapped to a CCS agent. The *state machine extraction algorithm* follows:

(1) Start at the root state.

(2) Consider all possible input events, as well as the possibility of no input.

(3) Update the output drivers with any new transactions.

(4) Advance the clock to the next transaction.

(5) Conduct the appropriate output events. Simultaneous outputs are translated as concurrent rather than simultaneous.

(6) Remove the transpired transactions from the drivers.

(7) Characterize the new state by the residual transactions on the drivers.

(8) Return to (2) and repeat until the entire behavior is expanded.

The algorithm mimics the VHDL simulation cycle, collecting the sequence of states and events as it does so. By so mimicking the simulation cycle, it faithfully captures the relative order of events intended by the VHDL model—except that simultaneous events are translated as merely concurrent. Since existing VHDL simulators can manage the states, events and transactions of the simulation cycle, it is clearly feasible to assume the same capability can be incorporated into a VHDL-to-CCS translator.

The algorithm characterizes states by the content of the signal drivers. The root state is *quiescent*, having no transactions scheduled. Applied to the VHDL model of the process within *H*, the results of this algorithm appear in the following Tables:

Table 5-1. Expansion of *H0*

state	H0 O(0) P(0)	
input	i	j
update drivers	O(0,1@1Δ) P(0,1@2Δ)	O(0,1@1Δ) P(0)
advance clock	Δ	Δ
output	$\bar{o}$	$\bar{o}$
new state	H1 O(1) P(0,1@1Δ)	H0 O(1) P(0)

Starting at the root state, here called *H0*, the outputs are given arbitrary starting values of '0'. Though the transitional semantics of CCS does not assign level values, the algorithm must track level values to properly maintain the drivers of O and P. Two branches of input behavior exit state *H*, the left-hand branch for the arrival of *i* and the right-hand branch for *j*. The possibility of no input need not be considered for a quiescent state such as *H*. The arrival of *i* and *j* simultaneously is also not considered. Since CCS cannot model this simultaneous arrival of inputs, one accepts this as a limitation of the translator.

For each behavioral branch, the drivers are updated and expressed in the following format:

<center><name>(<current value>, <transaction>, <transaction>, …)</center>

where each transaction is of the form

<center><new value>@<time></center>

After the drivers are updated the clock then advances to the next scheduled transaction. In both branches this increment is one delta, and the resulting event is on O in both cases, so an output $\bar{o}$ occurs. The residual drivers then characterize the new states. The right-hand branch has no transactions remaining, and it is recognized to be the root state *H0*. The fact that O now has current value of '1' instead of '0' is immaterial to the CCS transitional semantics, so this branch of behavior need not be further expanded.[3] For the left-hand branch, the pending transaction on P is brought one delta nearer in time, and the resulting state called *H1* is characterized by the one transaction on P scheduled for the next delta.

---

[3] This is only true when *all* the assignments are in the pseudo-transitional style. In general the translator *will* need to expand behavior until both the *level values* as well as the drivers match some previous state.

The CCS behavior of state *H0* can now be derived directly from this table, where the actions in parallel columns are connected by Choice, and the actions flowing down a single column are connected sequentially:

$$H0 \overset{def}{=} i.\overline{o}.H1 + j.\overline{o}.H0 \tag{5-10}$$

State *H1* merits further expansion:

Table 5-2. Expansion of *H1*

state	H1 $O(1)$ $P(0,1@1\Delta)$		
input	*i*	*j*	*none*
update drivers	$O(1,0@1\Delta)$ $P(0,1@1\Delta,1@2\Delta)$	$O(1,0@1\Delta)$ $P(0,1@1\Delta)$	$O(1)$ $P(0,1@1\Delta)$
advance time	Δ	Δ	Δ
output	$(\overline{o}\,|\,\overline{p}\,)$	$(\overline{o}\,|\,\overline{p}\,)$	$\overline{p}$
new state	**H2** $O(0)$ **P(1,1@1Δ)**	**H0** $O(0)$ $P(1)$	**H0** $O(1)$ $P(1)$

State *H1* is not quiescent. Having one pending transaction on P, the possibility of an event on P ahead of further inputs must also be considered. Hence there are three branches to consider. For the left-hand branch, transactions to both O and P are posted. The updated driver for P has two transactions to the same value, but at different times (1Δ and 2Δ). According to the VHDL semantics for *inertial* signals, when two transactions on a signal are to the same level value, both remain on the driver (Bhasker, 1999:98). The second transaction may be ineffective in producing a real signal change in P. Such transactions must be maintained, however, in case an intervening event changes the value of P. When the clock advances one delta, transactions for both O and P are encountered, and the simultaneous events are translated as the concurrent output ($\overline{o}\,|\,\overline{p}$). One residual

transaction remains on P's driver, and the new state is called *H2*. For the center branch, $(\bar{o} \,|\, \bar{p})$ is again generated, and the resulting state matches *H0* under transitional semantics. The right hand branch, receiving no inputs by the time the clock advances, simply emits the $\bar{p}$ and arrives at the quiescent state *H*. The CCS model for *H1* is thus

$$H1 \stackrel{def}{=} i.(\bar{o} \,|\, \bar{p}).H2 + j.(\bar{o} \,|\, \bar{p}).H0 + \bar{p}.H0 \qquad (5\text{-}11)$$

H2 is now expanded:

Table 5-3. Expansion of *H2*

state	H2 O(0) P(1,1@1Δ)		
input	*i*	*j*	*none*
update drivers	O(0,1@1Δ) P(1,~~1@1Δ~~,0@2Δ)	O(0,1@1Δ) P(1,1@1Δ)	O(0) P(1,1@1Δ)
advance time	Δ	Δ	Δ
output	$\bar{o}$	$\bar{o}$	*none*
new state	H1 O(1) P(1,0@1Δ)	H0 O(1) P(1)	H0 O(1) P(1)

The analysis for *H2* is straightforward in the center branch. The left-hand branch, however, has a curiosity. Here again P has transactions posted at one delta and two deltas. This time the transactions conflict. When the assignment is to a *different* level value, the first transaction is deleted, per the semantics of inertial signals (Bhasker, 1999:97). Hence, when $\bar{o}$ is emitted, the center branch arrives at a state with one transaction pending on P. This state is recognized to be identical to *H1* under transitional semantics. The right hand branch also shows a curiosity. When no inputs are received, the transaction on P transpires, but the event driving P to '1' is ineffective since P is

already at '1'. No actual transition occurs, and *H2* decays to *H* without receipt of input or generation of output.

$$H2 \stackrel{def}{=} i.\overline{o}.H1 + j.\overline{o}.H0 + H0 \qquad (5\text{-}12)$$

Once the process rooted at *H0* has been completely expanded the algorithm terminates. Since the initial model *H* had but one process, the target model *H* is identified with this one process: $H \stackrel{def}{=} H0$.

In summary, one now has is a mechanical means to produce a CCS state machine (target model) from a VHDL process. This algorithm expanded the single process within *H* rather quickly since all the output delays are deltas.   It could become extremely busy if one of the delays were finite, say, 1 nanosecond.   One might have to entertain the possibility of new inputs arriving every delta, of which there are an infinite number in the space of a nanosecond. Such intractability forces the designer to consider whether inputs will realistically arrive every delta, and, if not, to properly annotate his VHDL with assert statements, tests, etc., to properly limit the code in accordance with the system it purports to model.

Consider now initial models for the behavioral two-token buffers *FF* and *GG*. The reference models for these buffers admit nondeterminism due to the race between the generation of the first output and the receipt of the second input.   The environment controls the arrival of the input, but the model controls the emission of the output. CCS is indefinite about output emission, allowing it to occur anytime in the near or distant future. One can model such indefinite emission in VHDL by implementing a random number generator to assign delays, but it is unnatural and contrived to do so. Normally, the VHDL model uses typical or worst case delays. These delays, arising from solid-state circuitry, are certainly orders of magnitude less than the decades or centuries that the CCS model allows. Thus for the initial models, fixed delays will suffice.

Now a two-token buffer can accept *two* inputs without generating an output, but not *three*. In the following VHDL initial model the third output will be disallowed with an **assert** statement in the body of the main process.

```
-- Initial Model for Two-token Behavioral Buffer FF

entity FF is
    ( I : in bit; O : out bit )
end FF;

architecture BEHAVIOR of FF is
    constant DELAY1, DELAY2 :  time;
    signal STATE : (EMPTY_0, HALF_0, FULL_0,
                    EMPTY_1, HALF_1, FULL_1) := EMPTY_0;
begin
    process(I)
    begin
        assert (not(STATE = EMPTY_0 or STATE = EMPTY_1)
                                    and I'event
              );
        case STATE is
            when EMPTY_0  =>
                O <= 1 after DELAY1;
                STATE <= HALF_0, EMPTY_1 after DELAY1;
            when HALF_0 =>
                O <= 1 after DELAY2;
                STATE <= FULL_0, HALF_1 after DELAY2;
            when EMPTY_1 =>
                O <= 0 after DELAY1;
                STATE <= HALF_1, EMPTY_0 after DELAY1;
            when HALF_1 =>
                O <= 0 after DELAY2;
                STATE <= FULL_1, HALF_0 after DELAY2;
        end case;
    end process;
end BEHAVIOR;
```

DELAY1 and DELAY2 are constants of type **time**. Their values are not given here, but would be specified by the designer. An internally declared STATE signal tracks state information. STATE must be a VHDL *signal*, and not a *variable*, because a state change must be a scheduled future transaction that can be retracted.

This time, for variety, explicit level-signal assignments are used instead of the pseudo-transitional assignments of the single-token models. A STATE signal is also used. STATE takes on six possible values. Normally, one expects a two-token buffer to have three states: *empty*, *half-full* and *full*. With O receiving level assignments, the two possible values for O double the state possibilities.

The state machine extraction algorithm executes just as well with level assignments:

Table 5-4. Expansion of *FF0*

state	**FF0** STATE(**EMPTY_0**) O(0)
input	*i*
update drivers	STATE(EMPTY_0,HALF_0@Δ,EMPTY_1@DELAY1) O(0, 1@DELAY1)
advance clock	Δ
output	*none*
new state	**FF1** STATE(HALF_0,EMPTY_1@DELAY1) O(0, 1@DELAY1)

Table 5-5. Expansion of *FF1*

state	**FF1** STATE(HALF_0,EMPTY_1@DELAY1) O(0, 1@DELAY1)	
input	*I*	*none*
update drivers	STATE(HALF_0,FULL_0@Δ,HALF_1@DELAY2) O(0, 1@DELAY2)	STATE(HALF_0,EMPTY_1@DELAY1) O(0, 1@DELAY1)
adv. clock	Δ	DELAY1
output	*None*	$\overline{o}$
new state	**FF2** STATE(FULL_0,HALF_1@DELAY2) O(0, 1@DELAY2)	**FF3** STATE(EMPTY_1) O(1)

The state machine behavior can continue to be expanded. Since the signal STATE contains all necessary state information, the system-generated state labels *FF_x* are unneeded, and the six possible values of STATE can be used for state labels. This yields the following target model:

$$FF \stackrel{def}{=} EMPTY\_0 \stackrel{def}{=} i.HALF\_0$$

$$HALF\_0 \stackrel{def}{=} i.FULL\_0 + \overline{o}.EMPTY\_1$$

$$FULL\_0 \stackrel{def}{=} \overline{o}.HALF\_1$$

$$EMPTY\_1 \stackrel{def}{=} i.HALF\_1$$

$$HALF\_1 \stackrel{def}{=} i.FULL\_1 + \overline{o}.EMPTY\_0$$

$$FULL\_1 \stackrel{def}{=} \overline{o}.HALF\_0 \tag{5-13}$$

The initial and target models for *GG* are similar to those of *FF* with transitions on P simultaneous with those on O (in the initial model) and concurrent (in the target model):

```
-- Initial Model for Two-token Behavioral Buffer GG

entity GG is
   ( I : in bit; O, P : out bit )
end FF;

architecture BEHAVIOR of GG is
   constant DELAY1, DELAY2 :  time;
   signal STATE : (EMPTY_0, HALF_0, FULL_0,
                   EMPTY_1, HALF_1, FULL_1) := EMPTY_0;
begin
   process(I)
   begin
      assert (not(STATE = EMPTY_0 or STATE = EMPTY_1)
                               and  I'event       );
       case STATE is
          when EMPTY_0  =>
             O <= 1 after DELAY1;
             P <= 1 after DELAY1;
             STATE <= HALF_0, EMPTY_1 after DELAY1;
```

```
            when HALF_0 =>
               O <= 1 after DELAY2;
               P <= 1 after DELAY2;
               STATE <= FULL_0, HALF_1 after DELAY2;
            when EMPTY_1 =>
               O <= 0 after DELAY1;
               P <= 0 after DELAY1;
               STATE <= HALF_1, EMPTY_0 after DELAY1;
            when HALF_1 =>
               O <= 0 after DELAY2;
               P <= 0 after DELAY2;
               STATE <= FULL_1, HALF_0 after DELAY2;
        end case;
      end process;
   end BEHAVIOR;
```

$$GG \stackrel{def}{=} EMPTY\_0 \stackrel{def}{=} i.HALF\_0$$

$$HALF\_0 \stackrel{def}{=} i.FULL\_0 + (\overline{o} \,|\, \overline{p}).EMPTY\_1$$

$$FULL\_0 \stackrel{def}{=} (\overline{o} \,|\, \overline{p}).HALF\_1$$

$$EMPTY\_1 \stackrel{def}{=} i.HALF\_1$$

$$HALF\_1 \stackrel{def}{=} i.FULL\_1 + (\overline{o} \,|\, \overline{p}).EMPTY\_0$$

$$FULL\_1 \stackrel{def}{=} (\overline{o} \,|\, \overline{p}).HALF\_0 \tag{5-14}$$

In general, the algorithm translates one VHDL process into one CCS agent. The initial models given thus far have used single-process behavioral architectures. Indeed, there can be two or more processes within a behavioral architecture. When that happens a state machine is generated for each separately, and Parallel Composition connects them, since coexisting processes are considered parallel occurrences in VHDL. If such processes are totally independent, having no common input in their sensitivity lists, then the simple Parallel Composition suffices. If processes share a common sensitive signal, this cannot be directly modeled in CCS since multiple connections to a single port are not allowed. As discussed above, this limitation is imperfectly overcome by using a *FORK* element to split the signal.

$$FORK \overset{\text{def}}{=} i.\overline{o} \, . \, \overline{p} \, .FORK \qquad\qquad (5\text{-}15)$$

Now using the notation [[*V*]] to denote "the CCS translation of VHDL construct *V*," one has that

```
process(I,J)

      ...

  end process;

  process(I,K)

      ...

  end process;
```

translates to

$$(\quad FORK[m1/o,m2/p] \mid [[\texttt{process(I,J)}...;]][m1/j]$$

$$\mid [[\texttt{process(I,K)}...;]][m2/k] \ )\backslash\{m1,m2\} \qquad (5\text{-}16)$$

Now *structural* VHDL models contain *components*, and those components will be treated just like multiple processes, *i.e.*, they are concurrent and, upon translation, are connected by Parallel Composition. Consider the initial models for the structural two-place buffers. First, consider *FPF*:

```
-- Initial Model for Two-token Structural Buffer FPF

entity FPF is
    port( I: in bit; O: out bit);
end FPF;

architecture STRUCTURE of FPF is
```

```
        component F
            port( I : in bit; O : out bit);
        end component;
        signal M : bit;
        begin
            U1:F port map(I=>I, O =>M);
            U2:F port map(I=>M, O=>O);
        end STRUCTURE;

configuration CFG_FPF of FPF is
        for STRUCTURAL
            for all: F use configuration WORK.F.BEHAVIOR;
            end for;
        end for;
end CFG_FPF;
```

Being a structural VHDL model, the initial model for *FPF* must contain an explicit configuration body to import models for the subcomponents *F*. Note how this configuration calls out the default configuration for *F*, which is known as "WORK.F.behavior" in this implementation.

This model also contains an internal signal *M*. *M* is not a state signal, but an internal node to which component pins are attached. The components within the model are concurrent and connected by Parallel Composition upon translation. The **port map**s for the *U1, U2* components translate directly into Relabeling functions in CCS. Any internally declared signal, such as *M* here, is restricted upon translation. Thus, the target model for *FPF* is

$$FPF \stackrel{def}{=} F[m/o] \mid F[m/i] \setminus \{m\} \tag{5-17}$$

The initial model for *GPG* is similar to *FPF*:

```
-- Initial Model for Two-token Structural Buffer GPG

entity GPG is
    port( I: in bit; O,P: out bit);
end GPG;
```

```
architecture STRUCTURE of GPG is
    component G
        port( I : in bit; O,P : out bit);
    end component;
    signal M : bit;
begin
    U1:G port map(I =>I, O=>open, P=>M);
    U2:G port map(I=>M, O=>O, P=>P);
end STRUCTURE;

configuration CFG_GPG of GPG is
    for STRUCTURAL
        for all: G use configuration WORK.G.behavior;
        end for;
    end for;
end CFG_GPG;
```

Within *GPG* resides an **open** port assignment for *U1*. This unconnected pin is destined to become an *extraneous* output upon translation. Unlike the VHDL model, which hides this unconnected pin, the CCS model maintains it as an explicit, though extraneous, output action.

$$GPG \stackrel{def}{=} (\ G[x/o, m/p] \mid G[m/i]\ )\backslash\{m\} \tag{5-18}$$

The time has now arrived to capture VHDL-to-CCS translation rules that target the "pure" version of CCS (with interleaving concurrency). The VHDL subset supported by the hypothetical translator is quite broad, since all the capability of a VHDL simulator is presumed for the state machine extractor. Indeed, a reasonable way to build such a translator would use modify an existing VHDL analyzer and simulator as a front end. Thus, explicit and implicit processes are supported. Implicit and literal sensitivity lists are supported, as well as other process initiation schemes such as the **wait** statement. Both inertial and transport delay are supported, as well as both transitional and level

value assignments. All control structures such as **case**, **if … then …else**, **block** and so forth are supported.

In the discussion, all signals were of either of type **bit**, or are of enumerated type. This suffices since more complex structures can be built from assemblies of bits. Modes **in** and **out** are the only modes supported by the translation. Other modes, such as **inout** and **buffer**, can be recast as **in** and **out** with a little effort. The **generate** statement, **packages**, and a host of other features are not directly supported, but these constructs are typically pre-elaborated and result in constructs that *are* supported.

5.4 VHDL to CCS Translation Rules

Ten rules governing VHDL-to-CCS translation are now presented. Of these, the most significant rule is the state machine extraction algorithm, Rule 9.

(1) Each VHDL *design unit* (entity-architecture-configuration triple) is translated into a unique, capitalized name according to the scheme:

Design Unit Name ::= *<Entity Name>_<Architecture Name>_<Configuration Name>*

This naming convention is possible because all three items are required to have a design unit. Lone entities do not convey enough information for translation. In cases where the VHDL analyzer would normally supply a default configuration, that configuration is literally called "Default" within the CCS name.[4]

(2) Each process within a design unit is translated into a unique capitalized CCS agent name according to the scheme:

---

[4] The automated naming schemes given here tend to yield verbose names. For brevity, these schemes were *not* followed in the examples. For example, *H0* in Table 5-1 would, under these rules, have received the more cumbersome designation of *H_Behavior_Default_0.*

Process Name ::= *<Design Unit Name>*_Process*<sequence number>*

For a lone process within a design unit, the "_Process*<sequence number>*" appendage is omitted.

(3) Each VHDL signal is translated into a unique lower-case CCS action name, with the design unit information appended to assure uniqueness:

signal name ::= [']*<design unit name (lower case)>*_*<vhdl signal name>*

A leading apostrophe is added for signals of mode **out**. Any apostrophe appearing in VHDL names is translated literally as "_tick_" to avoid confusion with the CCS apostrophe.

(4) Locally declared signals used behaviorally (not connected to component ports) are recognized as state signals and are translated into multiple CCS agents—one agent for each state value used. The naming scheme is

State Name ::= *<Design Unit Name>*_*<VHDL Local Signal Name>*_*<State Value>*

(5) State signals created by the state machine extraction algorithm (Rule 5) are given names according to the scheme:

State Name ::= *<Process Name>*_*<sequence number>*

(6) Locally declared signals appearing in *structural* models are internal signals recognized because they connect to component ports. Their translation is to actions. The naming convention for the action is

local action name ::= *<design unit name(lower case)>_<vhdl local signal name>*

When multiply connected, they are modeled as FORKs, with the FORK outputs receiving an "*_<sequence number>*" appendage added to the local action name. Upon translation, these names are added to the Restriction set of the translated structural agent that contains the FORK.

(7) Port maps translate into Relabeling functions. Thus $I => M$ becomes [m/i].

(8) A port assignment to **open** indicates an unconnected pin. The pin becomes an *extraneous* output and gets relabeled in CCS to avoid collision. It does *not* get hidden as it does in the VHDL model. The translator is again presumed to have an unlimited supply of spare names to handle these cases. The format of this name is a lower case output name with a prefixed apostrophe:

'*<design unit name>_<translator-supplied symbol>*

where the translator-supplied symbol is different than any other user-defined symbol in the design unit.

(9) Processes are translated to CCS state machines by means of the *state machine extraction algorithm*.

(1) Start at the root state.

(2) Consider all possible input events, as well as the possibility of no input.

(3) Update the output drivers with any new transactions.

(4) Advance the clock to the next transaction.

(5) Conduct the appropriate output events. Any simultaneous outputs are translated as concurrent.

(6) Remove the transpired transactions from the drivers.

(7) Characterize the new state by the residual transactions on the drivers.

(8) Return to (2) and repeat until the entire behavior is expanded.

(10a) Multiple processes within a design unit that have no sensitive signals in common become concurrent CCS agents joined by Parallel Composition.

```
process(I)
begin
    …
end process;

process(J)
begin
    …
end process;
```

translates into:

```
[[process(I)…;]] | [[process(J)…;]]
```

(10b) When multiple processes within a design unit share the same sensitive signals, they are translated into a Parallel Composition with *FORK* agents added to split the input for separate communication with each agent. The translation of

```
process(I,J)
begin
     process body 1>
end process;

process(I,K)
begin
   <process body 2>
end process;
```

is

([[process(I,J)...;]]][*mid1/i*]
  | [[process(I,K)]]][*mid2/j*]
  | FORK[*mid1/o,mid2/p*]
)\{*midl, mid2*}

where *mid1* and *mid2* are drawn from an unlimited supply of extra names that the translator is presumed to have.

(10c) The units or components instantiated in a structural model are combined by Parallel Composition upon translation. When such components share a common input signal, that signal is modeled as a FORK in the same manner as the multiply-sensitive signals in Rule 9.

Rules 10a, 10b and 10c are really special cases of a more general rule that can be applied to models that employ a mixed behavioral/structural style. Hence they are combined as Rule 10:

(10) A design unit is translated by identifying its CCS name with the Parallel Composition of its contained processes and components. *To wit*,

<Design Unit Name> $\overset{def}{=}$

    ([[process$_1$]]][f_1] | [[process$_2$]]][f_2] | ... | [[process$_k$]]][f_k]

$$| [[\text{component}_1]][g_1] \ | [[\text{component}_2]][g_2] \ | \ldots \ | [[\text{component}_m]][g_m]$$

$$| \text{FORK}_1 \qquad\quad | \text{FORK}_2 \qquad\quad | \ldots | \text{FORK}_n$$

$$)\backslash\{\text{internal signals and their FORKed branches}\}$$

The FORKs are created and inserted as necessary to split multiply-connected internal nodes. The Relabeling functions f_i *and* g_i reassign sensitive signal names (for processes) and port names (for components) to receive the FORK outputs. All internal signals are then restricted.

The translation of components represents a recursive application of Rule 10, since components are in turn design units themselves. The recursion terminates when purely behavioral design units (containing only processes) are encountered.

In summary, the first eight rules govern the translation of *names*. Rule 9 shows how to translate *processes*. Rule 10 shows how to translate *structures* recursively, until only processes are encountered.

Note how Rule 10 is be exceedingly liberal in its use of FORKs to support the pure CCS semantics. The resulting CCS code is ungainly and probably impractical for all but the simplest verifications. However, this necessity emphasizes the semantic distance between CCS and VHDL, and the "pure" CCS translator serves as a more telling example of how $\succeq_w$ can be used to verify a translator, which is shown in the next section. Following that, a translator targeting the broadcast version of CCS, which is semantically less distant from VHDL, will be introduced.

5.5 Preservation of Congruent Weak Conformance

Does the translator outlined in the previous section preserve congruent weak conformance? That is, given two VHDL design units E and F, where $E \succeq_w F$, can one

say that $[[E]] \succeq_w [[F]]$? The question is interesting, because the issue of $\succeq_w$ among *VHDL* design units has not been specifically addressed or defined. VHDL modelers do not normally think of design units as being compliant to one another. Rather, a design unit is considered compliant to the *test bench* that exercises it. The test bench is little more than a behavioral model that provides a set of stimuli and checks for appropriate responses. The test bench thus models the environment. If two design units pass the same test bench then they are considered interchangeable within the environment modeled, but not necessarily compliant to each other.

5.5.1 Congruent Weak Conformance for VHDL Models. For the purpose of verifying the translator, $\succeq_w$ between VHDL models will mean the same as it does for CCS models, *i.e.*, that the four transitional laws and the five construction restrictions are satisfied. Therefore, the various symbols and concepts employed must have meaning within the context of VHDL.

Plainly, VHDL design units have both input and output sorts. They are given explicitly in the **entity** declaration. From these, the extraneous input and output sorts can be easily determined.

Furthermore, translation Rule 9 shows that VHDL code has a state machine interpretation. One can speak of labeled transitions such as $E \xrightarrow{a}$ when the possibility exists that the next simulated transition is on a.

As for τ transitions, VHDL has none. Yet the various internal signals of a structural VHDL model, though hidden at the top level, are tracked and managed by the simulator. Therefore, one can assert that $E \xrightarrow{\tau}$ when a transition on one of these internal signals is immediately pending.

The notion of *maxoctset* must be modified somewhat for VHDL models. Since simultaneity is allowed, VHDL maxoctsets may include paths with simultaneous output transitions.   In that event, the implementation is permitted to either duplicate the simultaneity, *or* perform the actions in some sequence.   Thus, if O and P have

simultaneous output events, then the derived maxoctset is {O.P, P.O, O&P} where '.' is borrowed from CCS to indicate sequential occurrence and where '&' denotes simultaneous occurrence.

5.5.2 Compliance Example Revisited. Thus $\succeq_w$ can be defined over VHDL just as it was for CCS agents. To illustrate, return to the BCD decoder example of Chapter 3. The BCD decoder had specification *S* had two implementations *I* and *J*. Possible VHDL models for these three agents appear in Appendix D in the level-signal modeling style.

The specification model *S* begins with an assertion that some input combinations are illegal. A large **case** statement then responds to the content of the inputs and posts transactions on all output drivers. Most of these transactions do not result in real transitions, since eight of the outputs will retain the same value from the previous state. Like the CCS model for *S*, only two of the outputs will experience a real transition. Since no explicit delay is specified, both transitions occur at the next delta. They are simultaneous, not simply concurrent. Thus, *S* has within its derivation tree, ten maxoctsets of the form {O.P, P.O, O&P}.

Not surprisingly, implementation *I* resembles *S* except for the addition of six outputs, and the lack of an assertion that bans certain input combinations. Once again, no delay is specified, and *I* implements the maxoctsets of *S* with the simultaneous option of the form O&P. The set {O10, O11, O12, O13, O14, O15} constitutes the *extraneous* output sort $\overline{Extr}\,(I,S)$, and transitions at these additional pins are tolerated. These transitions constitute relative taus, but produce no instability since they are guarded by input transitions.

J, on the other hand, has explicit delays associated with its output transactions. As in a real circuit, these delays are all different, and true simultaneity will not occur. Thus, for each maxoctset of *S*, *J* implements one of the sequential options and never the simultaneous option.

One can apply the translation rules to the VHDL models for *S*, *I* and *J*. Now the VHDL code is very "busy" in that it posts transactions to *every* output driver upon *every* change of input. Most of these transactions do not survive the translation, however. Since Rule 9 mimics the VHDL simulation engine, only the true transitions appear in the CCS target model. Thus, the CCS target models will be identical to the models given in Chapter 3, except that the names will be much more verbose. The agent corresponding to *S* will be called *S_Behavior_Default* and its states will be called *S_Behavior_Default_0*, *S_Behavior_Default_1*, and so forth. The action labels will be *s_behavior_default_a*, '*s_behavior_default_o1*, and so forth. Manual translations of the *S*, *I* and *J* design units appear in Appendix E.

In passing, it must be stated that the direct production of a CCS pair $[[S]] \succeq_w [[I]]$ from a VHDL pair $I \succeq_w S$ is an ideal that will be imperfectly realized. CCS supports full encapsulation of specification requirements as a state machine, with forbidden sequences simply omitted. Models are directly compared for conformance in a manner analogous to equivalence checking. VHDL must use **asserts** to prohibit certain sequences and, more likely, specification **asserts** will decorate the implementation model itself— there will be no separate specification *model*. This practice resembles the model checking approach. Of course, VHDL test benches are models that do check specification requirements, but they are not specification models in the same sense, just simple conduits for test vectors. One does not use a test bench as a placeholder in a system model, to be replaced by an implementation model. So the presentation of two VHDL models that share $\succeq_w$ will be rare.

5.5.3 Proof that $\succeq_w$ is Preserved. One is now assured that $\succeq_w$ has meaning when applied to VHDL models. One must now prove that whenever $E \succeq_w F$ then $[[E]] \succeq_w [[F]]$ necessarily follows. To establish this implication, one must prove two things: (1) that the translation rules preserve the five construction laws CS, PEA, ESP, COR and BR, and

(2) that the translated models are weakly conformant, *i.e.*, $[[E]] \succeq_w [[F]]$ (single underline).

Proposition 5-1. The translator's renaming function for VHDL design units and signals (Rules 1, 3, 6 and 8) is an injection.

Proof. Each VHDL design unit is identified with a unique entity-architecture-configuration triple. This unique information is carried forward in the translation, so the renaming of VHDL *design units* to CCS *agents* is injective. *Signal names*, whether internal or external, can be reused by different design units, but their scope is limited to that design unit. The VHDL analyzer must maintain such scoping information, so, in effect, each signal has a design unit tag that insures uniqueness. The *<design unit name>* prefix added in Rules 3 and 6 maintains that uniqueness in the target models. The renamed **open** port assignment of Rule 8 is unique by the assumption that the translator has an unlimited supply of spare names to secure this uniqueness. ☐

Since the renaming function is an injection, it is bijective with its own image. Therefore one can speak of the *reverse translation* $[[X]]^{-1}$ of CCS agents and actions, as long as the reverse translation is not applied to the *additional* signals (Rules 6 and 8) and agents (Rules 2, 5 and 9) that the translation generates.

Proposition 5-2 establishes that actions whose order of occurrence is fixed in the VHDL model will be sequential in the CCS model. Actions whose order of occurrence is either ambiguous or simultaneous on the VHDL side is concurrent in the CCS translation.

Proposition 5-2. *The translator preserves any absolute ordering of actions.*

Proof.

- *Rules 1* through *8*. These rules affect only the assignment of names. They do not redefine sequence of actions.

- *Rule 9*. As a faithful mimic of the VHDL simulation cycle, the state machine extraction algorithm preserves the same ordering and concurrency of events expressed in the VHDL code. Simultaneous events become ambiguously ordered (*i.e.*, concurrent), but any event occurring strictly before or after a simultaneity occurs strictly before or after the corresponding concurrency after translation.

- *Rule* (*10*). There are two cases to consider: (1) independence (no common inputs) and (2) dependence on common input signals.

  Case 1. Processes and components that share no common inputs can proceed independently. Thus, they are concurrent. Any interleaving of the events of each process can occur that is consistent with the behavior of each process individually. The Parallel Composition in the target model mirrors that concurrency.

  Case 2. When a group of processes and components share common, sensitive inputs, they progress independently except at points where they await a common input. After receipt of that input, they proceed independently to the next common input. CCS cannot activate two agents at the same time as VHDL does. However, insertion of the *FORK* agent causes the components or processes to wait at the same point for the common sensitive signal, after which they can continue independently.

  ◻

Proposition 5-3. *The translation rules preserve the CS assumption.*

Proof. Since, by assumption, CS holds for the pre-translated VHDL models—there can be no unguarded relative taus in the initial models. One must consider whether, as a

result of translation, (1) guarded relative taus can move to unguarded positions, (2) an unguarded action can become an extraneous output action by renaming and (3) the extraneous action created by Rule 8 can become unguarded.

- *Case 1.* Proposition 5-2 guarantees that absolute orderings among events are preserved, so such moves are not possible.

- *Case 2.* The renaming function of design units and signals is injective by Proposition 5-1, so the membership of signals(actions) in the sorts of design units (agents) does not change. Hence their *extraneous* or *non-extraneous* nature cannot change.

- *Case 3.* The component output left **open** constitutes a hidden action in the VHDL model, which becomes an extraneous output upon translation. Again, the CS assumption insures that the VHDL hidden action is guarded; and Proposition 5-2 prevents the resulting extraneous output in the CCS model from migrating to an unguarded position.

<div style="text-align: right;">◻</div>

Proposition 5-4. *The translator preserves the PEA, ESP and COR conditions.*

Proof. The renaming of actions, as an injection, preserves the $\subseteq$ and $\in$ relationships within the definitions of PEA, ESP and COR. ◻

Proposition 5-5. *The translator preserves the BR condition.*

Proof. All relabelings (port maps) on the VHDL side are bijective by assumption. The translator renaming function is bijective to its own image. Hence the Relabelings resulting from translated port maps are bijective, since the bijection of a bijection is bijective. It remains to show that Relabelings introduced by the Rule 10 preserve BR.

These functions rename process and component inputs to communicate with the branches of the introduced FORKs. Yet these branches bear new and unique names per the naming scheme of Rule 6, so BR is preserved. ◻

Proposition 5-6. *The translator preserves the $\succeq_w$ construction laws.*

Proof. Propositions 5-3, 5-4 and 5-5. ◻

Having shown that the translator preserves the construction laws, one must now establish that $[[E]] \succeq_w [[F]]$. This can be shown by coinductive proof that the relation $S \equiv \{\,([[E]], [[F]]) : E \succeq_w F\,\}$ is a weak conformation up to $\succeq_w$. First, one must show that the translator preserves $\approx$ and also preserves all maxoctsets. One must also show that the reverse translation $[[\ldots]]^{-1}$, when it exists, preserves these properties.

Proposition 5-7. The translator and its inverse preserve observational equivalence $\approx$.

Proof:

- *Translator.* Show that $S \equiv \{\,(\,[[P]], [[Q]]\,) : P \approx Q\,\}$ is a bisimulation up to $\approx$.

  Whenever $[[P]] \stackrel{\alpha}{\Rightarrow} R'$ then $P \stackrel{\kappa}{\Rightarrow} P'$ where $[[\kappa]] = \alpha$ and $[[P']] = R'$.

  Since $P \approx Q$ then $Q \stackrel{\kappa}{\Rightarrow} Q' \approx P'$.

  Hence $[[Q]] \stackrel{\alpha}{\Rightarrow} [[Q']]$ and clearly, $(\,[[P']], [[Q']]\,) \in S$.

- *Inverse translator.* Define $S \equiv \{\,(P, Q) : [[P]] \approx [[Q]]\,\}$. The proof is similar.

  ◻

Proposition 5-8. The translator and its inverse preserve maxoctsets.

Proof:

- *Translator.*

Let M be a maxoctset of F converging at F'. That is, $\forall m \in M$, $F \overset{m}{\Rightarrow} \approx F'$.

Hence $\forall [[m]] \in [[M]]$, $[[F]] \overset{[[m]]}{\Rightarrow} \approx [[F']]$ since the translator preserves $\approx$.

Thus $[[M]]$ is at least an octset of $[[F]]$, converging on $[[F']]$.

Trial Hypothesis. Assume $[[M]]$ is not a maxoctset.

Then $[[F]]$ has maxoctset N where

$$N = \{[[m]].n : m \in M, n \in \overline{\mathcal{A}}([[F]])^+ \text{ and } [[F']] \overset{n}{\Rightarrow} S''\}.$$

Yet if $[[F']] \overset{n}{\Rightarrow} \approx S''$ then $F' \overset{[[n]]^{-1}}{\Rightarrow} \approx F''$ since $[[\ldots]]^{-1}$ preserves $\approx$.

Hence $F \overset{m}{\Rightarrow} \overset{[[n]]^{-1}}{\Rightarrow} \approx F''$ $\forall [[m]].n \in N$.

Hence $[[N]]^{-1}$ is an octset of F and M cannot be a maxoctset.

$\Rightarrow\Leftarrow$

Thus $[[M]]$ is a maxoctset of $[[F]]$ converging at $[[F']]$.

- *Inverse translator.* Similar.

$\blacksquare$

Proposition 5-9. $\mathcal{S} \equiv \{ ([[E]], [[F]]) : E \succeq_w F \}$ *is a weak conformation up to* $\succeq_w$.

Proof:

- *LSIT'.* Let $[[F]] \overset{\alpha}{\Rightarrow} S'$. Then $F \overset{\sigma}{\Rightarrow} F'$ where $[[\sigma]] = \alpha$ and $[[F']] = S'$.

  By LSIT, $E \overset{t}{\Rightarrow} E' \succeq_w F'$.

  Since absolute order among actions is preserved (Proposition 5-2), then

  $$[[E]] \overset{u}{\Rightarrow} [[E']] \text{ where } u = [[t]].$$

  Since $E' \succeq_w F'$ then clearly, $[[E']] \, \mathcal{S} \, [[F']]$.

- *LSO'.* Let $[[F]]$ have a maxoctset N converging at S'.

  Then F has maxoctset $[[N]]^{-1}$ converging at some F', where $F' = [[S']]$.

  E implements $[[N]]^{-1}$ with t:

  $$E \overset{t}{\Rightarrow} E' \succeq_w F' \text{ with } t \restriction \mathcal{A}(F) = n \in N.$$

  Thus

$$[[E]] \overset{[[t]]}{\Rightarrow} [[E']], \, [[t]] \upharpoonright \mathcal{A}([[F]]) = [[n]] \in [[N]]$$

and clearly, $[[E']] \, \mathcal{S} \, [[F']]$.

- *LII', LIOT'.* Similar to LSIT'.

$\blacksquare$

Proposition 5-10. $\succeq_w$ is preserved by the translator.

Proof: Propositions 5-6 and 5-9. $\blacksquare$

Thus, the translator defined by Rules 1 to 10 preserves congruent weak conformance, and the translator is validated.

5.6 Translation to Broadcast CCS

The translator was able to bridge the semantic gap between VHDL and CCS while preserving $\succeq_w$ in the process. Since pure CCS does not support multiple fan-out, the translator had to insert FORK agents into the target code. This necessity to use FORKs to mimic broadcast communication is a major contributor to the semantic gap between VHDL and CCS. Furthermore, the resulting target code is very limited in the types of verifications it can support. Using FORKs to model interconnect corresponds to the *delay-insensitive* asynchronous design model, where the delay associated with interconnect is completely unspecified. Designs verified under the delay-insensitive assumption are very robust, but achieving a working design under delay-insensitive constraints is very hard. Only a very limited number of circuits can be designed using delay-insensitive techniques.

More practical asynchronous designs use the *speed-independent* model. Interconnect wiring is assumed to have minuscule delay compared to the delay of components. Thus a signal propagates essentially simultaneously along the branches of

any multiply connected node. This is fairly consistent with VHDL modeling, where wire alone has no delay, though inconsistent with pure CCS modeling.

To model speed-independent designs, Stevens added transition rules to CCS for supporting broadcast communication (Stevens, 1994: 180-5). The *conjunction* operator, $|_c$, is a modification of Parallel Composition, and allows a single output to handshake with more than one input action. Five transition rules give the semantics of $|_c$. Of particular interest is **Conj₄**:

$$Conj_4 \frac{E \xrightarrow{\ l\ } E' \quad F \xrightarrow{\ \bar{l}\ } F'}{E \mid_c F \xrightarrow{\ \bar{l}\ } E' \mid_c F'}$$

where $\bar{l}$ is understood to be an output. Compare this with Milner's **Com₃**:

$$Com_3 \frac{E \xrightarrow{l} E' \quad F \xrightarrow{\bar{l}} F'}{E \mid_c F \xrightarrow{\tau} E' \mid_c F'}$$

For Milner's rule, an action/coaction pair collapses to τ, so no other agents can synchronize on it. Stevens rule collapses it to the output $\bar{l}$, ready to communicate with additional input actions. Thus, a single output can drive multiple inputs, thereby modeling broadcast communication.

Though Stevens did not name this modified CCS, here it will be called *BCCS* (*Broadcast CCS*) here to distinguish it from Milner's.

A VHDL-to-BCCS translator need not resort to the insertion of FORKs. Thus the revised translator is simpler. Rules 6 and 10 are modified to eliminate the insertion of FORKs.

(6') Locally declared signals appearing in *structural* models are internal signals recognized because they connect to component ports. Their translation is to actions. The naming convention for the action is

local action name ::= <design unit name(lower case)>_<vhdl local signal name>

A local action can connect one input to multiple outputs without FORKs.

(10') A design unit is translated by identifying its CCS name with the Conjunction of its contained processes and components. *To wit,*

<Design Unit Name> $\overset{def}{=}$

$\quad$ ($\;$ [[process$_1$]] $\quad$ | [[process$_2$]] $\quad$ | ... | [[process$_k$]]

$\quad\quad$ | [[component$_1$]] | [[component$_2$]] | ... | [[component$_m$]]

$\quad$)\{internal signals }

Thus the VHDL-to-BCCS translator is somewhat simpler. Does this translator also preserve $\succeq_w$?

Proposition 5-11. $\succeq_w$ is preserved by the VHDL-to-BCCS translator.

Proof: Examine Proposition 5-1 through 5-10 as they apply to the new translator. Proposition 5-1 still holds as stated. The proof of 5-2 is simpler since it is uncomplicated by FORKs. The proofs of the remaining propositions, which depend on 5-1 and 5-2, are unmodified, so the VHDL-to-BCCS preserves $\succeq_w$ as well. $\quad\quad\blacksquare$

5.7 Conclusion

This chapter addressed the use of congruent weak conformance $\succeq_w$ to verify a set of transformations from one modeling language to another. The verification of such transformations is particularly challenging when the underlying semantics of the two languages differ. Loss of information is inevitable in such translation. However $\succeq_w$, which models the designer's desire for safe substitution, needs to be preserved in the course of a useful translation.

In this chapter, VHDL-to-CCS translation was examined under the light of $\succeq_w$. First, the various semantic differences between VHDL and CCS were duly noted. Next, some small, but interesting agent models were presented to elucidate those differences and to serve as a basis for further study of the translation process. These models were selected to exhibit the salient feature of $\succeq_w$ by incorporating examples of extraneous actions, hidden actions, and maxoctsets.

The models were presented in three groups. First, the *reference* models served simply to present the behavior under discussion. Though given in CCS, the reference models were not intended to be translated or to be the result of translation. Instead, the true objects of translation were the *initial* models, given in VHDL. The initial models were *inspired* by, but not formally derived from, the reference models. The *target* models, however, were CCS models mechanically derived from the initial models by a hypothetical translator.

The translator consisted of ten rules. The most important rule mapped the VHDL *process* construct to a CCS *agent*. Parallel Composition then combined agents that represented parallel-acting processes and components in the VHDL models. Other rules defined the assignment of agent and action names upon translation. In section 5.4, the translator was then shown to preserve $\succeq_w$ during the course of translation. Thus, the translator preserved safe substitution, and therefore the translator was verified.

A second translator that targeted the more practical *Broadcast CCS* was also discussed. Since Broadcast CCS has a communication semantics closer to VHDL, the second translator is less complex. This translator was also verified by proof that it preserves $\succeq_w$.

In summary, this chapter demonstrated the use of $\succeq_w$ to verify transformations between systems of unlike semantics. Language translators are but one example of such transformations. Synthesis and extraction tools such as those used in integrated circuit design are also transformation systems, and should be amenable to verification by $\succeq_w$.

VI. Conclusion

This chapter summarizes the dissertation, lists the original contributions of the present research, and provides recommendations for future research.

6.1 Summary

This dissertation started with intuitive, informal notions of compliance. In the end, a property called *congruent weak conformance* that captured these intuitive notions was defined, formally developed, and used to verify translations between incompatible semantic systems.

Chapter 1 gave a brief introduction to the problem of compliance, and ended with a diagram (Figure 1-1) showing that the desired property is probably not an equivalence.

Chapter 2 presented the prior art. In particular, Chapter 2 presented various formal equivalences that have been defined over the process algebra CCS. However, since a conforming implementation can exhibit behaviors in excess of its specification, equivalences were seen to be insufficient to the task. Therefore, various process-ordering relations from the literature were also presented, with their shortcoming noted as well.

To extract all the intuitive aspects of conformance, Chapter 3 provided a simple example of a specification (a BCD converter) and a conforming implementation (an appropriately wired demultiplexor). The intuition gained from the example was then formalized into four transition rules. These rules defined a family of relations called *weak conformations*. Some formal results were proven for weak conformations. However, the weak conformations were only precursor relations, and needed refinement to produce the ultimate desired property.

These refinements were presented in Chapter 4. The first refinement, *weak conformance* ($\succeq_w$), was presented as the largest of the weak conformations. Formal

results governing $\succeq_w$ were proven, and the property held promise. Yet it was not possible to show that $\succeq_w$ is fully substitutable, *i.e.* it is not a *congruence*. Hence, additional refinement was needed. This refinement consisted of five constructional restrictions that govern the building of specification and implementation models. These restrictions were shown to be very reasonable constraints that do nothing more than codify good design intent. Congruent weak conformance $\succeq_w$ was then defined as $\succeq_w$ with the additional refinement that the five design restrictions must be observed. Congruent weak conformance was then proven to be a congruent partial order, or *precongruence*. This established $\succeq_w$ as the final desired property: the loosest known model of conformance that provides for safe substitution.

To apply $\succeq_w$ to a practical problem, Chapter 5 investigated a hypothetical VHDL-to-CCS translator. This type of translator is challenging to build because the VHDL and CCS semantics are incompatible, and information must necessarily be lost in the course of the translation. For verifications upon translated models to be valid, some appropriate property must be preserved by the translator. Since the designer's art can be characterized as the search for an implementation to substitute for a specification, then *safe substitution* (*i.e.* $\succeq_w$) is the property that must be preserved.

The hypothetical translator was summarized as a set of ten transformations. Each transformation was then shown to preserve $\succeq_w$. This in effect verified the VHDL-to-CCS translator. A translator from VHDL to a more practical *Broadcast* CCS was also verified.

The objects of this research were five-fold:

1. Determine the characteristics of a compliant device with respect to its specification. Study the expected behavior of an implementation in response to specified input, output and hidden action. Conversely, note any reverse

obligations of the specification to implemented input, output and hidden action.

2. Incorporate this intuition into the formally defined property of *congruent weak conformance* as a binary relation over processes. Make this formal property as "loose" as possible such that it admits all appropriate implementations and allows maximum design flexibility.

3. Derive formal results for congruent weak conformance and related properties. Prove that congruent weak conformance is partial order. Prove also that congruent weak conformance is fully substitutable in all contexts, is a valid model of safe substitution, and is indeed a *congruence*.

4. Outline the transformations necessary to create a semantic link from VHDL to CCS.

5. Show that such transformations are valid by proof that they preserve congruent weak conformance, thus allowing the more powerful verifications of CCS to accrue to VHDL models.

These objects have been accomplished.

6.2 Contributions

This research has yielded several original contributions: (1) *local confluence*, (2) the concept of a *maxoctset*, (3) the transitional laws that define the *weak conformations*, (4) *relative stability*, (5) the five model-construction restrictions, (6) the relation $\succeq_w$, and (7) a methodology for verifying transformations between systems of unlike semantics.

6.2.1 Local confluence. Local confluence is a looser notion than the classical confluence. Local confluence identifies areas of a transition graph that exhibit confluent-

like behavior in the absence of a global confluence. With local confluence, one can identify and exploit behavioral options offered by a specification model.

6.2.2 Maxoctsets. The *maxoctset* denotes local confluence among outputs and hence represents output options offered by the specification. Maxoctsets are the *largest* such areas that can be identified. Being the largest, they represent the least restriction possible restriction on the designer in implementing specified output actions.

6.2.3 Weak conformations. Four transitional laws characterize the weak conformations. These laws provide the greatest flexibility in implementation by

(1) abstracting hidden action in a manner similar to weak bisimulation.

(2) allowing the implementation to engage in unspecified behavior in the unreachable state space in a manner similar to logic conformance,

(3) providing for additional I/O pins in the implementation that do not block specified behavior,

(4) allowing maximum exploitation of output concurrency through the use of maxoctsets

As the largest weak conformation, weak conformance, $\succeq_w$, unites every pair of process agents that share the transitional laws.

6.2.4 Relative stability. The notion of *stability* has been generalized to *relative stability* in recognition that unguarded extraneous outputs can play the same destabilizing role as unguarded taus in creating unstable models.

6.2.5 Model construction restrictions. Five design restrictions pertinent to the construction of models have been identified. These restrictions are an embodiment of consistent design intent. Adherence to these restrictions is a prerequisite to achieving safe substitution.

6.2.6 Congruent weak conformance. The congruent weak conformance property

$\succeq_w$ is the final process relation derived in this research. As a refinement of $\succeq_w$, it enjoys the design flexibility of the weak conformations. By incorporating the five model construction restrictions, it also models safe substitution . Hence, $\succeq_w$ is a congruence, and this has been shown by extensive proof.

6.2.7 Transformation verification methodology. If translation or other transformations between systems with incompatible semantics is attempted, then safe substitution (*i.e.* congruence) must be preserved, even when other information is lost. As the loosest known precongruence, $\succeq_w$ thus forms a useful tool to verify such transformations. Such usage has been illustrated by the verification of two hypothetical VHDL-to-CCS translators.

6.3 Recommendations for Future Work

More work can always be done. In the course of the present research, several interesting topics became manifest as possible follow-on efforts. These topics are:

(1) Axiomatization of $\succeq_w$.

(2) Automated $\succeq_w$ tool.

(3) Implemented VHDL-to-CCS translator.

(4) Verification of translators.

(5) Verification of synthesis tools.

6.3.1 Axiomatization of $\succeq_w$. Congruent weak conformance was defined in terms of four transitional laws and five design restrictions. Thus, to prove that $A \succeq_w B$ one must ultimately show that A and B satisfy these laws and restrictions. An alternate formalization of $\succeq_w$ by means of axioms may be possible. A few primitive pairs of agents would be assumed to observe $\succeq_w$, and this would yield a set of axioms. The proof that

$A \succeq_w B$ would then be a theorem to be derived from the axioms. Observational equivalence = was axiomatized in this way (Milner, 1989:160-9).

6.3.2 Automated $\succeq_w$ tool. For the present research it sufficed to use $\succeq_w$ as a manual proof tool to verify transformations. For any two agents A and B, whether or not $A \succeq_w B$ must be proven manually at the present time. However, an automated tool to establish $\succeq_w$ between two agents would be a great aid to designers and logisticians. Furthermore, if an axiomatization of $\succeq_w$ is achieved, then an extant automated theorem proving tool could be used in this role.

Guidance for producing an automated $\succeq_w$ tool is now offered. This tool must determine two things: (1) that a weak conformation exists between two agents and (2) that the five construction restrictions are obeyed. Both tasks require knowledge of the extraneous sorts, with (2) requiring a more detailed knowledge of the extraneous sorts of any component agents down to the purely behavioral level. Thus the $\succeq_w$ tool will likely accomplish the following tasks:

(1) Extract the input and output sorts of each agent as well as the sorts for each component agent. As a compromise, the tool will extract syntactic sorts only, since the determination of semantic sorts is undecidable. Having settled on syntactic sorts only, a straightforward lexical analysis of agent expressions will then suffice to accomplish this task.

(2) Calculate the extraneous sorts of each agent and any component agents. These extraneous sorts will be easily derived by set difference.

(3) Check for violations of the five construction laws. The analysis will halt and report if any such violation is found.

(4) Use an appropriate algorithm to determine if a weak conformation exists. Such an algorithm has not yet been invented. No doubt it will be similar to existing algorithms that are used to determine the existence of a bisimulation between agents (Cleaveland, 1989; Fernandez, 1989; Stevens 1994, 194-195). However, any weak conformation algorithm will probably be more complex than any of the bisimulation algorithms, owing to the greater complexity of the weak conformation laws over bisimulation laws.

Once the $\succeq_w$ tool is built, test cases will then be needed to validate the tool. Obviously, both behavioral and structural agent pairs that are known to share the $\succeq_w$ relation will need to be submitted as validation tests. Equally important, though, are pairs that are expected to fail $\succeq_w$. Failure cases must be constructed to contain violations of each of the five construction restrictions. In addition, tests that challenge each aspect of the conformation laws need to be constructed. Examples of failure cases that should appear in any validation suite include:

(1) interleaving inputs (*specified* as well as *extraneous* inputs) that take the implementation to illegal behavior.

(2) interleaving outputs (specified and extraneous) that take the implementation to illegal behavior.

(3) maxoctsets illegally implemented. For example, the implementing string may contain the proper output actions, but in an order that is unspecified, *i.e.*, that order of actions is missing from the maxoctset.

(4) specifications having actions extraneous to the implementation. These should result in immediate violation of LSIT and LSO.

(5) implementations that lead to illegal behaviors triggered by an input action $\xrightarrow{\gamma}$,

when the specification can or cannot perform an immediate $\overset{\gamma}{\Rightarrow}$.

(6) agent pairs that are not relatively stable.

(7) constructions that promote extraneous actions to specified actions, using Prefix, Choice and Parallel Composition.

(8) constructions that attempt to synchronize on extraneous actions.

(9) relabeling functions that are not bijective.

(10) non-idle extraneous outputs illegally restricted.

These above guidelines will hopefully aid the production and testing of a future congruent weak conformance checking tool.

6.3.3 Implemented VHDL-to-CCS translator. Chapter 5 provides the outline of a VHDL-to-CCS translator. The obvious next step is to build and verify such a translator. Such a translator would allow the bisimulation-based verifications possible within CCS to accrue to VHDL models. Hence such a translator would be a good design aid.

Guidance for producing such a translator has been given already, since an explicit listing of the translation rules is given in Chapter 5. The most difficult rule to realize will be the state machine extraction algorithm. Yet that algorithm mimics the VHDL simulation cycle, and implemented VHDL simulators abound. Thus, a likely way to implement the translator may be by modification of existing VHDL analyzers and simulators, with a new "back-end" targeted to output CCS state machines in place of an explicit event-based simulation.

6.3.4 Verification of translators. Similarly, other existing translators, as well as newly introduced translators, could and should be verified using $\succeq_w$. The bigger the semantic gap between the initial language and its target, the more useful such a verification could be.

6.3.5 Verification of synthesis tools. In the design world, transformational systems abound. Though one does not think of them as translators, the class of computer-aided design tools called *synthesis* tools are, in fact, transformational systems. They perform transformations between design language models, schematics, netlists and even silicon layout. There is often a semantic gap involved in synthesis. For example, logic that can be expressed in a language model may not have an exact equivalent within the component library of a particular technology, and a component of different functionality may be substituted. Furthermore, once an actual layout is generated, certain physical parameters (resistance, delay time, etc.) become instantiated, and these may have an effect on the desired functionality. These parameters often have to be "back-annotated" into the original model so the functionality can be rechecked.   Thus, synthesis is an example of inter-semantic translation. There is a change or loss of information in the process. Again, safe substitution must be preserved. This suggests that $\succeq_w$ be used to verify such synthesis tools.

6.4 Concluding Remarks

The objects of this research were met. Intuitive notions of conformance were captured formally. The resulting property, congruent weak conformance, was then shown to be a congruence, and therefore a correct model of safe substitution. Congruent weak conformance was then successfully used to verify transformations between systems of unlike semantics. Thus congruent weak conformance was shown to be a useful verification tool.

Appendix A

Strong Conformation

For *strong conformations*, τ is treated as an output. The implementation must match explicit τ actions in the specification, though it is free to add more of its own. Specified τ actions are treated like outputs since, like outputs, the environment cannot control their emission. The maxoctset concept must be modified to include τ actions amidst the outputs. Call this a *maxtoctset* (with an extra 't' for 'tau'). LSO becomes "LSOT" and LSIT, which loses its τ role, becomes "LSI." LII and LIOT are unmodified.

For LSOT, the implementation selects some string *s* from the maxtoctset, where *s* consists of both outputs *and* τ actions. This sequence of outputs and taus must be faithfully implemented, and the specified τ actions cannot be deleted in the implementation, though additional τ actions and extraneous outputs, may be added. $t \restriction \overline{\mathcal{A}}(S) = s$ no longer captures the desired relationship between *t* and *s*, since *s* has embedded taus. $t \restriction (\overline{\mathcal{A}}(S) \cup \{\tau\}) = s$ does not work either, due to the extra taus that *t* may add. Therefore *s* has to be expressed as $\beta_0.\beta_1....\beta_n$ where the β_i range over output and τ.

Definition A-1. A binary relation on processes, $S \subseteq \mathcal{P} \times \mathcal{P}$, is a *strong conformation* if $\forall \alpha \in \mathcal{A}(S), \forall \beta \in \overline{\mathcal{A}}(I) \cup \{\tau\}, \forall \gamma \in \mathcal{A}(S)$, *I S S implies* the following four laws:

Law of Specified Input (LSI)
Whenever $S \overset{\alpha}{\Rightarrow} S'$ then $\exists t \in (\mathcal{A}(S) \cup \overline{\mathcal{Extr}}(I,S))^*$ such that

    (1) $I \overset{t}{\Rightarrow} I'$

    (2) $t \restriction \mathcal{A}(S) = \alpha$

    (3) $I' S S'$

Law of Specified Output or Tau (LSOT)

Let X be a maxtoctset of S. $\exists s = \beta_0.\beta_1....\beta_n \in X$ (where $\beta_i \in \mathcal{A}(S) \cup \{\tau\}$) and

$\exists t \in (\overline{\mathcal{A}}(I) \cup \{\tau\})^+$ such that

    (1) $S \xrightarrow{s} S'$

    (2) $I \xRightarrow{t} I'$

    (3) $t \upharpoonright \overline{\mathcal{A}}(S) \cup \{\tau\} = \tau^*.\beta_0.\tau^*.\beta_1.\ \tau^*.\beta_2.\ \tau^*.....\tau^*.\beta_n.\ \tau^*$

    (4) $I' \ \mathcal{W} \ S'$

Law of Implemented Input (LII)

Whenever $I \xrightarrow{\gamma} I'$ and $S \xRightarrow{\gamma}$ then

    (1) $S \xRightarrow{\gamma} S'$

    (2) $I' \ S \ S'$

Law of Implemented Output or Tau (LIOT)

Whenever $I \xrightarrow{\beta} I'$ and $\delta \equiv \beta \upharpoonright \overline{\mathcal{A}}(S)$ then

    (1) $S \xRightarrow{\delta} S'$

    (2) $I' \ S \ S'$

Appendix B.

Lengthy proofs

Proof of Proposition 3-9.

For $P \, \mathcal{V} Q \, \mathcal{W} R$ let $R \xrightarrow{\alpha} R'$ for $\alpha \in \mathcal{A}(R) \cup \{\tau\}$.

- Applying LSIT on $Q \, \mathcal{W} R$ yields some $s \in (\mathcal{A}(R) \cup \overline{\mathcal{E}xtr}(Q,R))^*$ such that

  $Q \xRightarrow{s} Q'$, $s \upharpoonright \mathcal{A}(R) = \hat{\alpha}$, $Q' \, \mathcal{W} R'$.

- Rewrite $Q \xRightarrow{s} Q'$ as $Q \xRightarrow{u} Q_1 \xrightarrow{\hat{\alpha}} Q_2 \xRightarrow{v} Q'$ where $u, v \in \overline{\mathcal{E}xtr}(Q,R)^*$ and any τ actions in

  $\xRightarrow{s}$ are subsumed by $\xRightarrow{u}$ and $\xRightarrow{v}$. ($\xrightarrow{\hat{\alpha}}$ represents the empty transition $\xrightarrow{\varepsilon}$ for $\alpha = \tau$.)

- There are two cases for string u: (1) $u = \varepsilon$ and (2) $u \neq \varepsilon$.

  Case 1. $P \Rightarrow P_1 \, \mathcal{V} Q_1$ by LSE.

  Case 2. u is a *specified output* to be implemented by P under LSO.

  $\therefore \exists r \in \overline{\mathcal{A}}(P)^+$ such that

  $P \xRightarrow{r} P_1$, $r \upharpoonright \overline{\mathcal{A}}(Q) =_{\text{conf}} u$, $P_1 \, \mathcal{V} Q_1$.

- Since $P_1 \, \mathcal{V} Q_1$ and $Q_1 \xrightarrow{\hat{\alpha}} Q_2$ there are two cases for $\hat{\alpha}$: (1) $\hat{\alpha} \neq \varepsilon$ and (2) $\hat{\alpha} = \varepsilon$.

  Case 1. Apply LSO. $\exists x \in [\mathcal{A}(Q) \cup \overline{\mathcal{E}xtr}(P,Q)]^+$ such that

  $P_1 \xRightarrow{x} P_2$, $x \upharpoonright \mathcal{A}(Q) = \hat{\alpha}$, $P_2 \, \mathcal{V} Q_2$.

  Case 2. Apply LSE.

  $P_1 \Rightarrow P_2$, $Q_1 \Rightarrow Q_2$, $P_2 \, \mathcal{V} Q_2$.

- Performing the same case analysis on v that was done on u yields:

  $P_2 \xRightarrow{y} P' \, \mathcal{V} Q'$ for some $y \in \overline{\mathcal{A}}(P)^*$ where $y \upharpoonright \overline{\mathcal{A}}(Q) =_{\text{conf}} v$.

- $\therefore P \xRightarrow{t} P'$ where $t = r.x.y$ and $P' \, \mathcal{V} \mathcal{W} R'$.

It remains to be shown that $t \upharpoonright \mathcal{A}(R) = \hat{\alpha}$.

- Both r and y are composed of *output* strings. Their projections onto *input* sort $\mathcal{A}(R)$ are empty.

- $\therefore t \upharpoonright \mathcal{A}(R) = r.x.y \upharpoonright \mathcal{A}(R) = x \upharpoonright \mathcal{A}(R)$.

- By LSIT on $P_1 \lor Q_1$ one knows that $x \upharpoonright \mathcal{A}(Q) = \hat{\alpha}$.

- Since $\mathcal{A}(R) \subseteq \mathcal{A}(Q)$ then $x \upharpoonright \mathcal{A}(R) = x \upharpoonright \mathcal{A}(Q) \upharpoonright \mathcal{A}(R) = \hat{\alpha} \upharpoonright \mathcal{A}(R) = \hat{\alpha}$.

$\blacksquare$

Proof of Lemma 4-12.

First show that every maxoctset of $(S \mid T)$ can be represented with respect to *some* $s_i.t_j$. Then show that *every* such $s_i.t_j$ defines a maxoctset of $(S \mid T)$.

- Let M be a maxoctset of $(S \mid T)$ with respect to r.

- Write $r =_{\text{conf}} s.t$ where s and t are the actions provided to r by S and T, respectively, with the symbols appearing in r in the same order they appear in s and t, though they are intermixed in r.

- Thus $S \overset{s}{\Rightarrow} S'$, $T \overset{t}{\Rightarrow} T'$ and $(S \mid T) \overset{r}{\Rightarrow} (S' \mid T')$.

- By definition, $\forall r_i \in M, (S \mid T) \overset{r_i}{\Rightarrow} \approx (S' \mid T')$.

- Similar to r, let $r_i =_{\text{conf}} s_i.t_i$ such that $S \overset{s_i}{\Rightarrow} \approx S'$ and $T \overset{t_i}{\Rightarrow} \approx T'$.

- Note that $s_i =_{\text{conf}} s$ and $t_i =_{\text{conf}} t$.

- $\therefore s$ defines an *octset* for S, as does t for T.

- *Trial Hypothesis*. Assume one of these octsets is not a *maxoctset*. W.l.o.g. let it be the octset of S defined by s.

- Then S must then have a maxoctset M_S' with respect to $s.s''$ for some $s'' \neq \varepsilon$.

- $\forall s' \in M_S' : S \overset{s'}{\Rightarrow} \approx S''$.

- $\forall r' =_{\text{conf}} s.s''.t$ where $(S \mid T) \overset{r'}{\Rightarrow}$ one has $(S \mid T) \overset{r'}{\Rightarrow} \approx (S'' \mid T')$.

- $\therefore \{r' =_{\text{conf}} s.s''.t : (S \mid T) \overset{r'}{\Rightarrow}\}$ is an octset of $(S \mid T)$ and $\therefore M$ cannot be a maxoctset.

$\Rightarrow\Leftarrow$

Now show that *every* such $s_i.t_j$ defines a maxoctset of $(S|T)$.

- Let $M_{ij} = \{ r : (S \mid T) \overset{r}{\Rightarrow}$ and $r =_{\text{conf}} s_i.t_i \}$.

- $\forall x \in M_{ij}$ write $x =_{\text{conf}} s.t$ for some $s \in Y_i$, $t \in Z_j$.

- Since Y_i and Z_j are maxoctsets one derives that $\forall s : S \overset{s}{\Rightarrow} \approx S'$ and $\forall t : T \overset{t}{\Rightarrow} \approx T'$.

- $\therefore \forall x \in M_{ij} : (S \mid T) \overset{x}{\Rightarrow} \approx (S' \mid T')$ and hence M_{ij} is (at least) an *octset* of $(S \mid T)$.

- *Trial Hypothesis.* Assume that M_{ij} is not a *maxoctset* of $(S \mid T)$.

- Then $(S \mid T)$ has some maxoctset with respect to $s_i t_j.x'$ where $x' \neq \varepsilon$.

- Let $x' =_{\text{conf}} s'.t'$ where s', t' are the contributions to x', in sequence, from S and T.

- *At least one* of $s', t' \neq \varepsilon$. W.l.o.g. let $s' \neq \varepsilon$.

- $\forall r =_{\text{conf}} s_i.s'$ such that $S \overset{r}{\Rightarrow}$, one has $S \overset{r}{\Rightarrow} \approx S''$. $\therefore S$ has an octset with respect to $s_i.s'$.

- $\therefore Y_i$ cannot be a maxoctset of S.

$\Rightarrow\Leftarrow$

Proof of Proposition 4-13.

Given $I \,\mathcal{W}\, S$ and $J \,\mathcal{W}\, T$, construct the relation

$$S \equiv \{ \ ((I \mid J), (S \mid T)) : I \,\mathcal{W}S, J \,\mathcal{W}T \ \}$$

and show that S is a weak conformation.

- *LSIT.* Let $(S \mid T) \overset{\alpha}{\rightarrow}$ where $\alpha \in \mathcal{A}(S \mid T)$. There are two cases to consider:

  (1) α is a visible action or an explicit τ emitted by *one* of the two agents.

  (2) α is a τ arising from communication *between* the two agents.

  Case 1. Assume w.l.o.g. that $S \overset{\alpha}{\rightarrow} S'$. Apply LSIT on $I \,\mathcal{W}S$ yielding:

  $$I \overset{t}{\Rightarrow} I'$$

  $$t \restriction \mathcal{A}(S) = \hat{\alpha}$$

$I' \, \mathcal{W} \, S'$.

Hence, whenever $(S \mid T) \xrightarrow{a} (S' \mid T)$ then

$$(I \mid J) \xRightarrow{t} (I' \mid J)$$

Furthermore,

$$t \upharpoonright \mathcal{A}(S \mid T) = \hat{\alpha}$$

by an argument analogous to Proposition 4-11 (LSIT).

Finally,

$$(I' \mid J) \, \mathcal{S} \, (S' \mid T)$$

since $I' \, \mathcal{W} \, S'$ and $J \, \mathcal{W} \, T$.

Case 2. Assume w.l.o.g. that $S \xrightarrow{a} S'$ and $T \xrightarrow{\bar{a}}$ where a is an input and $\bar{a}$ an output.

(Note that $\bar{a}$ is a participant in some maxoctset of S and, though it appears first in *some* member string of that maxoctset, it may not appear first in the member string that is implemented by J.)

First apply LSO to $J \, \mathcal{W} \, T$:

$$T \xRightarrow{s_1} \xrightarrow{\bar{a}} \xRightarrow{s_2} T'$$

$$J \xRightarrow{t_1} \xrightarrow{\bar{a}} \xRightarrow{t_2} J'$$

$$t_1 \upharpoonright \overline{\mathcal{A}}(T) = s_1, \quad t_2 \upharpoonright \overline{\mathcal{A}}(T) = s_2$$

$$J' \, \mathcal{W} \, T'.$$

Then apply LSIT to $I \, \mathcal{W} \, S$:

$$S \xrightarrow{a} S'$$

$$I \xRightarrow{r_1} \xrightarrow{a} \xRightarrow{r_2} I'$$

$$r_1 \upharpoonright \mathcal{A}(S) = \varepsilon, \quad a \upharpoonright \mathcal{A}(S) = a, \quad r_1 \upharpoonright \mathcal{A}(S) = \varepsilon$$

$$I' \, \mathcal{W} \, S'.$$

Combining the results of LSO and LSIT under Parallel Composition:

$$(S \mid T) \xRightarrow{s_1} \xrightarrow{\tau_a} \xRightarrow{s_2} (S' \mid T') \setminus$$

$$(I \mid J) \xRightarrow{u_1} \xrightarrow{\tau_a} \xRightarrow{u_2} (I' \mid J')$$

B-4

where $u_1 =_{conf} r_1.t_1$ and $u_2 =_{conf} r_2.t_2$.

Now $r_1, r_2 \in \overline{\mathcal{E}xtr}(S)$ and $\therefore$ by PEA, $r_1, r_2 \in \overline{\mathcal{E}xtr}((S \mid T))$ as well.

PEA also prevents the promotion of extraneous outputs in t_1 and t_2. Hence

$$t_1 \upharpoonright \overline{\mathcal{A}}(S \mid T) = t_1 \upharpoonright \overline{\mathcal{A}}(T) = s_1$$

$$t_2 \upharpoonright \overline{\mathcal{A}}(S \mid T) = s_2.$$

Thus,

$$u_1.\tau_a.u_2 \upharpoonright \overline{\mathcal{A}}((S \mid T)) = s_1.s_2$$

snd clearly

$$(I' \mid J') \ S \ (S' \mid T').$$

- *LSO.*

    Let M be a maxoctset of $(S \mid T)$.

    By Lemma 4-12, $\exists$maxoctsets Y of S and Z of T such that

    M is a maxoctset with respect to $y.z$ for some $y \in Y_i$ and $z \in Z_j$.

    By LSO, $\exists s \in Y$ such that

    $$S \overset{s}{\Rightarrow} S', \ I \overset{u}{\Rightarrow} I', \ u \upharpoonright \overline{\mathcal{A}}(S) = s, \ I' \ \mathcal{W} \ S'.$$

    Similarly, $\exists t \in Z$ such that

    $$T \overset{t}{\Rightarrow} T', \ J \overset{v}{\Rightarrow} J', \ v \upharpoonright \overline{\mathcal{A}}(T) = v, \ J' \ \mathcal{W} \ T'.$$

    $\therefore$ $(S \mid T) \overset{s}{\Rightarrow} (S' \mid T) \overset{t}{\Rightarrow} (S' \mid T')$ and similarly, $(I \mid J) \overset{u}{\Rightarrow} \overset{v}{\Rightarrow} (I' \mid J')$.

    Since $(S \mid T) \overset{s}{\Rightarrow} \overset{t}{\Rightarrow}$ and $s.t =_{conf} y.z$ then $s.t \in M$.

    PEA guarantees that no extraneous outputs in u or v are promoted.

    $\therefore$ $u \upharpoonright \overline{\mathcal{A}}((S \mid T)) = u \upharpoonright \overline{\mathcal{A}}(S) = s$ and similarly, $v \upharpoonright \overline{\mathcal{A}}((S \mid T)) = t$.

    Hence $u.v \upharpoonright \overline{\mathcal{A}}((S \mid T)) = s.t$.

    Clearly $(I' \mid J') \ S \ (S' \mid T')$.

    $\therefore$ $u.v$ is a valid implementation by $(I \mid J)$ of $s.t \in M$.

- *LII.*

    Let $(I \mid J) \overset{\gamma}{\not\to}$ for $\gamma \in \mathcal{A}((S \mid T))$. W.l.o.g. assume $I \overset{\gamma}{\not\to} I'$.

    LII requires that:

$S \overset{\gamma}{\Rightarrow} S'$, $I' \mathcal{W} S'$.

Thus: $(S \mid T) \overset{\gamma}{\Rightarrow} (S' \mid T)$,

$\qquad (I \mid J) \overset{\gamma}{\rightarrow} (I' \mid J)$ and

$\qquad (I' \mid J) \, S \, (S' \mid T)$.

- *LIOT.*

  Let $(I \mid J) \overset{\beta}{\rightarrow} P'$ where $\beta \in \overline{\mathcal{A}}(I \mid J) \cup \{\tau\}$. There are two cases to consider:

  $\qquad$ (1) β is an output or explicit τ emitted by one of the components.

  $\qquad$ (2) β is a τ arising from communication between the components.

  Case 1. W.l.o.g. let $I \overset{\beta}{\rightarrow} I'$. Thus $P' = (I' \mid J)$

  $\qquad$ By LIOT on $I \mathcal{W} S$ one has

  $\qquad\qquad S \overset{\delta}{\Rightarrow} S'$, $\delta \equiv \beta \upharpoonright \overline{\mathcal{A}}(S)$, $I' \mathcal{W} S'$.

  $\qquad$ If $\beta \notin \overline{\mathcal{A}}(S)$ then $\beta \notin \overline{\mathcal{A}}(S \mid T))$ by the PEA condition.

  $\qquad \therefore \delta \equiv \beta \upharpoonright \overline{\mathcal{A}}(S) = \beta \upharpoonright \overline{\mathcal{A}}(S \mid T)$.

  $\qquad$ Hence one has:

  $\qquad\qquad (I \mid J) \overset{\beta}{\rightarrow} (I' \mid J)$

  $\qquad\qquad (S \mid T) \overset{\delta}{\Rightarrow} (S' \mid T)$

  $\qquad\qquad \beta \upharpoonright \overline{\mathcal{A}}((S \mid T)) = \delta$

  $\qquad\qquad (I' \mid J) \, S \, (S' \mid T)$.

  Case 2. W.l.o.g. let $I \overset{a}{\rightarrow} I'$ and $J \overset{\bar{a}}{\rightarrow} J'$, where a is the input action.

  $\qquad (I \mid J) \overset{\tau_a}{\rightarrow} (I' \mid J')$ and thus $P' = (I' \mid J')$.

  $\qquad$ There are two cases on a. All other cases violate ESP.

  $\qquad\qquad$ (a) $a \in \mathcal{A}(S)$ and $\bar{a} \in \overline{\mathcal{A}}(T)$.

  $\qquad\qquad$ (b) $a \in \mathcal{Extr}(I,S)$ and $\bar{a} \in \overline{\mathcal{Extr}}(J,T)$.

  $\qquad$ *Case a.* First, apply LIOT to $J \mathcal{W} T$:

  $\qquad\qquad T \overset{\delta}{\Rightarrow} T'$, $\delta \equiv \bar{a} \upharpoonright \overline{\mathcal{A}}(T) = \bar{a}$, $J \overset{\bar{a}}{\Rightarrow} J'$, $J' \mathcal{W} T'$.

  $\qquad$ Now apply LII to $I \mathcal{W} S$:

  $\qquad\qquad S \overset{a}{\Rightarrow} S'$, $I' \mathcal{W} S'$, $(S \mid T) \overset{\tau_a}{\Rightarrow} (S' \mid T')$. $(I' \mid J') \, S \, (S' \mid T')$.

Case b. Both signals are extraneous to their respective specifications.

$$\therefore S \overset{\tau}{\Rightarrow} S', \ T{\Rightarrow}T', \ (S \mid T) \overset{\tau}{\Rightarrow} (S' \mid T').$$

$$\text{Again, } (I' \mid J') \ \mathcal{S} \ (S' \mid T').$$

<div align="right">◻</div>

Proof of Proposition 4-14.

It suffices to conduct the proof for a singleton Restriction set $\{c\}$. All others will yield to induction.

- Let $I \ \mathcal{W} S$ for some weak conformation $\mathcal{W}$.

- Define $S_C \equiv \{ (P\backslash\{c\}, Q\backslash\{c\}) : P \ \mathcal{W} Q \}$. Show that S_C is a weak conformation.

- *LSIT.*

   From LSIT on the base agents one derives

   $$S \overset{\alpha}{\Rightarrow} S', \ I{\overset{t}{\Rightarrow}}I', \ t{\restriction}\mathcal{A}(S) = \hat{\alpha}, \ I' \ \mathcal{W} S'.$$

   If $\alpha \in \mathcal{A}(S\backslash\{c\})$ then $\alpha \neq c$ and $\therefore S\backslash\{c\} \overset{\alpha}{\Rightarrow} S'\backslash\{c\}$.

   Now t contains no inputs other than a single $\alpha \neq c$.

   In accordance with COR, there are three possibilities for $\bar{c}$:

   $$(1) \ \bar{c} \notin \overline{\mathcal{A}} \ (I), \ \overline{\mathcal{A}} \ (S),$$

   $$(2) \ \bar{c} \in \overline{Idle} \ (I,S) \text{ and}$$

   $$(3) \ \bar{c} \in \overline{\mathcal{A}}(S).$$

   Case 1. $\bar{c} \notin \overline{\mathcal{A}} \ (I)$ and $\therefore \bar{c} \notin t$.

   Case 2. $\bar{c}$ cannot lie along any reachable path. Clearly t is a reachable path. $\therefore \bar{c} \notin t$.

   Case 3. $\bar{c} \in \overline{\mathcal{A}}(S)$. $\therefore \bar{c} \notin \overline{Extr}(I,S)$. All *outputs* in t come from $\overline{Extr}(I,S)$. $\therefore \bar{c} \notin t$.

   For all three cases $\bar{c} \notin t$. $\therefore I\backslash\{c\} \overset{t}{\Rightarrow} I'\backslash\{c\}$ and $(I'\backslash\{c\}, S'\backslash\{c\}) \in S_C$.

- *LSO.*

   Let M be a maxoctset of S with respect to s.

   There are two cases on $\bar{c}$: (1) $\bar{c} \in s$ and (2) $\bar{c} \notin s$.

Case 1. The Restriction blocks every $x \in M$ and the application of LSO to is moot.

Case 2. M remains whole, and $\therefore \forall x \in M : S \setminus \{c\} \overset{x}{\Rightarrow} S' \setminus \{c\}$.

  Let t be the implementation of M by I.

  Now $\bar{c} \notin t$.   Otherwise, $\bar{c} \in \overline{\mathcal{E}xtr}\,(I,S)$, a violation of COR.

  $\therefore I \setminus \{c\} \overset{t}{\Rightarrow} I' \setminus \{c\}$.

  Also, since $\bar{c} \notin t$ one has: $t \upharpoonright \mathcal{A}(S \setminus \{c\}) = t \upharpoonright \mathcal{A}(S) = s$.

  Clearly, $(\, I^{\wedge} \setminus \{c\}, S^{\wedge} \setminus \{c\} \,) \in \mathcal{S}_C$, so LSO is established.

- *LII and LIOT:* Similar.

$\blacksquare$

Proof of Proposition 4-17.

Define $\mathcal{R} \equiv \{\, (G\{I/X\},\ G\{S/X\}) : I \overset{def}{=} E\{I/X\},\ S \overset{def}{=} F\{S/X\}\ \}$.   Show that $\mathcal{R}$ is a weak conformation up to $\succeq_w$.   Once that is established then $G\{I/X\}\ \mathcal{R}\ G\{S/X\}$ implies that $G\{I/X\} \succeq_w G\{S/X\}$.   In particular, when $G\{X\} \equiv X$ one derives $I \succeq_w S$.   To show $\mathcal{R}$ to be a weak conformation up to $\succeq_w$ one must establish each of the "primed" laws.   The proof of each law is a coinduction on the structure of G, also known as *transistion induction* (Milner, 1989: Section 2.10).   The cases are $G \equiv X$ (recursive definition), $G \equiv \alpha.G1$, $G \equiv G1 + G2$, $G \equiv G1|G2$, $G \equiv G1 \setminus \{c\}$, $G \equiv G1[f]$ and $G \equiv C$ (a constant agent having no occurrences of X).

- *LSIT'.*

  To show, for $\alpha \in \mathcal{A}(G\{S/X\}) \cup \{\tau\}$:

    Whenever $G\{S/X\} \overset{\alpha}{\rightarrow} Q'$ then $G(I/X) \overset{t}{\Rightarrow} P' \succeq_w \mathcal{R} \succeq_w Q'$ where $t \upharpoonright \mathcal{A}(G\{S/X\}) = \hat{\alpha}$.

  $G \equiv X$.

    In this case $G\{S/X\} \equiv S$.

    Let $G\{S/X\} \equiv S \overset{\alpha}{\rightarrow} Q'$ and consider the inference that established this transition.

    It arises from application of **Con**, where the side condition is $S \overset{def}{=} F\{S/X\}$.

B-8

Hence, by a shorter inference, $F\{S/X\} \xrightarrow{\alpha} Q'$.

By coinduction, $F\{I/X\} \xRightarrow{t} P' \succeq_w \mathcal{R} \succeq_w Q'$ where $t \restriction \mathcal{A}(F\{S/X\}) = \hat{\alpha}$.

Now apply $E \succeq_w F$. $E\{I/X\} \succeq_w F\{I/X\}$.

Since $F\{I/X\} \xRightarrow{t} P'$ then $E\{I/X\}$ implements t with some u:

    t contains *at most* one input $\hat{\alpha}$ amidst extraneous ouputs. *i.e.* $t = t'.\hat{\alpha}.t'''$.

    $u = u'.u''.u'''$ arises by serial application of LSOS, LSIT and LSOS.

    $u' \restriction \overline{\mathcal{A}}(F\{I/X\}) =_{\text{conf}} t'$. $u'' \restriction \mathcal{A}(F\{I/X\}) = \hat{\alpha}$. $u''' \restriction \overline{\mathcal{A}}(F\{I/X\}) =_{\text{conf}} t'''$.

Hence $E\{I/X\} \xRightarrow{u} O' \succeq_w P'$.

But $G\{I/X\} \equiv I \stackrel{\text{def}}{=} E\{I/X\}$ so $G\{I/X\} \xRightarrow{u} O'$.

$u \restriction \mathcal{A}(F\{S/X\}) = \hat{\alpha}$.

$O' \succeq_w P' \succeq_w \mathcal{R} \succeq_w Q'$. That is, $O' \succeq_w \mathcal{R} \succeq_w Q'$.

Hence LSIT' is established for the case $G \equiv X$.

$G \equiv \alpha.G1$.

    $G\{S/X\} \equiv \alpha.G1\{S/X\} \xrightarrow{\alpha} G1\{S/X\}$ and $G\{I/X\} \equiv \alpha.G1\{I/X\} \xrightarrow{\alpha} G1\{I/X\}$

    Clearly, $G1\{I/X\} \mathrel{\mathcal{R}} G1\{S/X\}$.

$G \equiv G1 + G2$.

    Let $G\{S/X\} \xrightarrow{\alpha} Q'$. The transition arises from **Sum**.

    By a shorter inference, $G1\{S/X\} \xrightarrow{\alpha} Q'$ or $G2\{S/X\} \xrightarrow{\alpha} Q'$.

    W.l.o.g. assume the former.

    By coinduction, $G1\{I/X\} \xRightarrow{t} P' \succeq_w \mathcal{R} \succeq_w Q'$ for $t \restriction \mathcal{A}(G1\{S/X\}) = \hat{\alpha}$.

    $t \restriction \mathcal{A}(G\{S/X\}) = \hat{\alpha}$ since t contains no inputs but $\hat{\alpha}$.

    $G\{I/X\} \xRightarrow{t} P' \succeq_w \mathcal{R} \succeq_w Q'$ by **Sum**.

$G \equiv G1|G2$.

    Let $G\{S/X\} \xrightarrow{\alpha} Q'$. The transition arises from **Com1**, **2** or **3**.

    Com1.

        By a shorter inference, and w.l.o.g., $G1\{S/X\} \xrightarrow{\alpha} Q1'$.

        By coinduction $G1\{I/X\} \xRightarrow{t} P1' \succeq_w \mathcal{R} \succeq_w Q1'$ for $t \restriction \mathcal{A}(G1\{S/X\}) = \hat{\alpha}$.

$t \upharpoonright \mathcal{A}(G\{S/X\}) = \hat{\alpha}$ since t contains no inputs but $\hat{\alpha}$.

Thus $G\{I/X\} \overset{t}{\Rightarrow} P1'|G2\{I/X\}$ when $G\{S/X\} \overset{\alpha}{\rightarrow} Q1'|G2\{I/X\}$

It remains to show that $P1'|G2\{I/X\} \succeq_w \mathcal{R} \succeq_w Q1'|G2\{I/X\}$.

Write $P1' \succeq_w \mathcal{R} \succeq_w Q1'$ as $P1' \succeq_w P1'' \mathcal{R} Q1'' \succeq_w Q1'$.

$P1'' \mathcal{R} Q1''$ means $\exists H1\{X\}$ such that

$P1'' \equiv H1\{I/X\}$ and $Q1'' \equiv H1\{S/X\}$.

Set $H \equiv H1|G2$.

$P1'|G2\{I/X\} \succeq_w P1''|G2\{I/X\} \equiv H\{I/X\} \mathcal{R} H\{S/X\}$ and

$$H\{S/X\} \equiv Q1'|G2\{I/X\} \succeq_w Q1'|G2\{I/X\}$$

Thus $P1'|G2\{I/X\} \succeq_w \mathcal{R} \succeq_w Q1'|G2\{I/X\}$.

Com2. Similar.

Com3. By a shorter inference, $G1\{S/X\} \overset{a}{\rightarrow} Q1'$. $G2\{S/X\} \overset{\bar{a}}{\rightarrow} Q2'$.

W.l.o.g., a is an input action and $\bar{a}$ is its output coaction.

By coinduction, LSIT' applies to the former, and LSO' applies to the latter.

$G1\{I/X\}$ and $G2\{I/X\}$ implement with strings containing a and $\bar{a}$.

This situation was faced in the proof of Proposition 4-13, LSIT, Case 2.

The proof that:

$$(G1\{S/X\} \mid G2\{S/X\}) \overset{\tau_a}{\rightarrow} (Q1' \mid Q2')$$

$$(G1\{I/X\} \mid G2\{I/X\}) \overset{t''}{\Rightarrow} (P1' \mid P2')$$

$$t'' \upharpoonright \mathcal{A}(G1\{S/X\} \mid G2\{S/X\}) = \varepsilon$$

is analogous to that of Proposition 4-13, where

$G1\{S/X\}$ plays the role of S, $Q1'$ of S',

$G2\{S/X\}$ of T, $Q2'$ of T',

$G1\{I/X\}$ of I, $P1'$ of I',

$G2\{S/X\}$ of J and $P2'$ of T'.

By coinduction both

$$P1' \succeq_w \mathcal{R} \succeq_w Q1'$$

B-10

and

$P2' \succeq_w \mathcal{R} \succeq_w Q2'.$

Writing them as

$P1' \succeq_w H1'\{I/X\} \mathcal{R} H1'\{S/X\} \succeq_w Q1'$

$P2' \succeq_w H2'\{I/X\} \mathcal{R} H2'\{S/X\} \succeq_w Q2'$

one has

$(P1' \mid P2') \succeq_w (H1'\{I/X\} \mid H2'\{I/X\})$ by Proposition 4-13,

$(H1'\{I/X\} \mid H2'\{I/X\}) \mathcal{R} (H1'\{S/X\} \mid H2'\{S/X\})$ by definition of '$\mathcal{R}$'

$(H1'\{S/X\} \mid H2'\{S/X\}) \succeq_w (Q1' \mid Q2')$ by Proposition 4-13

from which follows:

$(P1' \mid P2') \succeq_w \mathcal{R} \succeq_w (Q1' \mid Q2').$

This proof that $\succeq_w \mathcal{R} \succeq_w$ applies to the derivatives of a composite agent follows a single scheme, as can by seen in the **Com1** and **Com3** cases immediately above. The scheme is as follows:

(1) Take the '$\succeq_w \mathcal{R} \succeq_w$' relation(s) of the non-composite derivatives.

(2) Introduce expression(s) H to fill in between '$\succeq_w$' and '$\mathcal{R}$'.

(3) Form a composite H.

(4) Invoke the proposition that states that the operator in question preserves weak conformation.

(5) Show that the composite deriviatives share '$\succeq_w$' with the composite H.

(6) Note that (composite $H\{I/X\}$, composite $H\{S/X\}$) $\in \mathcal{R}$.

(6) Combine results to show that the composite derivatives share '$\succeq_w \mathcal{R} \succeq_w$'.

This scheme will be frequently reused, and is now called the *composite H scheme*.

$G \equiv G1 \backslash \{c\}$.

$G\{S/X\} \equiv G1\{S/X\} \backslash \{c\} \xrightarrow{\alpha} Q'$.

The transition arises from application of **Res** with the side condition $\alpha \neq c$.

By a shorter inference $G1\{S/X\} \xrightarrow{\alpha} Q''$ and hence $Q' = Q'' \backslash \{c\}$.

By induction $G1\{I/X\} \xRightarrow{t} P'' \succeq_w \mathcal{R} \succeq_w Q''$ for $t \upharpoonright \mathcal{A}(G1\{S/X\}) = \hat{\alpha}$.

The case analysis on $\bar{c}$ in Proposition 4-14 applies here, and $\therefore \bar{c} \notin t$.

Hence $G1\{I/X\} \backslash \{c\} \xRightarrow{t} P'' \backslash \{c\}$

$P'' \backslash \{c\} \succeq_w \mathcal{R} \succeq_w Q'' \backslash \{c\}$ is shown by means of the composite H scheme.

$G \equiv G1[f]$.

$G\{S/X\} \equiv G1\{S/X\}[f] \xrightarrow{f(\alpha)} Q'[f]$

The transition arises by application of **Rel**.

By a shorter inference, $G1\{S/X\} \xrightarrow{\alpha} Q'$.

By induction, $G1\{I/X\} \xRightarrow{t} P' \succeq_w \mathcal{R} \succeq_w Q'$.

By **Rel** $G\{I/X\} \xRightarrow{f(t)} P'[f]$.

$P'[f] \succeq_w \mathcal{R} \succeq_w Q'[f]$ is shown by means of the composite H scheme.

$G \equiv C$.

$G\{I/X\} \equiv G\{S/X\}$ and the satisfaction of LSIT' is trivial.

- *LSO'*.

To show that $\forall$maxoctsets M of $G(S/X)$:

$G(I/X\} \xRightarrow{t} P' \succeq_w \mathcal{R} \succeq_w Q'$ where $G\{S/X\} \xRightarrow{s} Q'$ and $t \upharpoonright \overline{\mathcal{A}} (G\{S/X\}) = s$.

$G \equiv X$.

$G\{S/X\} \equiv S$ and $S \overset{def}{=} F\{S/X\}$.

$\forall m \in M$, whenever $S \xRightarrow{m} \approx Q'$ then $F\{S/X\} \xRightarrow{m} \approx Q'$ by shorter inferences.[1]

Hence $F\{S/X\}$ has a maxocset M.

---

[1] Since the transition rules derive atomic transitions, multiple applications are required to infer a string transition.

By coinduction, $F\{I/X\} \overset{t}{\Rightarrow} P' \succeq_w R \succeq_w Q'$ where $t \upharpoonright \overline{\mathcal{A}}(F\{S/X\} = s \in M$.

t defines a maxoctset M' for $F\{I/X\}$ by Proposition 3-6.

Apply $E \succeq_w F$. $E\{I/X\} \succeq_w F\{I/X\}$.

By LSO, $E\{I/X\}$ implements M' with some u where $u \upharpoonright \overline{\mathcal{A}}(F\{I/X\}) =_{conf} t$.

Hence $E\{I/X\} \overset{u}{\Rightarrow} O' \succeq_w P'$.

But $G\{I/X\} \equiv I \overset{def}{=} E\{I/X\}$ so $G\{I/X\} \overset{u}{\Rightarrow} O'$.

Since $u \upharpoonright \overline{\mathcal{A}}(F\{I/X\}) =_{conf} t$ then

$$u \upharpoonright \overline{\mathcal{A}}(F\{S/X\}) = u \upharpoonright \overline{\mathcal{A}}(F\{I/X\}) \upharpoonright \overline{\mathcal{A}}(F\{S/X\}) = s' =_{conf} s.$$

$s' \in M$ for the same reasons that $x \in X$ in the proof of Proposition 3-7.

Now $O' \succeq_w P' \succeq_w R \succeq_w Q'$ and hence $O' \succeq_w R \succeq_w Q'$.

$G \equiv \alpha.G1$.

Let $G\{S/X\} \equiv \alpha.G1\{S/X\}$ have maxoctset M with respect to m, terminating at Q'.

α must be an output action and must appear first in *every* string of M.

$G1\{S/X\}$ has maxoctset $M1$ with respect to m_1, where $m = \alpha.m_1$.

$M1$ also terminates at Q'.

By coinduction, $G1\{I/X\} \overset{t}{\Rightarrow} P' \succeq_w R \succeq_w Q'$ where $t \upharpoonright \overline{\mathcal{A}}(G1\{S/X\}) = s \in M1$.

Hence $G\{I/X\} \overset{\alpha.t}{\Rightarrow} P'$ and $G\{S/X\} \overset{s}{\Rightarrow} Q'$.

$\alpha.t \upharpoonright \overline{\mathcal{A}}(\alpha.G1\{S/X\}) = \alpha.s \in M$.

$P' \succeq_w R \succeq_w Q'$ is already established.

$G \equiv G1 + G2$.

When $G\{S/X\}$ has maxoctset M then by Lemma 4-10, $M = M1 \cup M2$ where

$M1$ is a maxoctset of $G1\{S/X\}$ and $M2$ is a maxoctset of $G2\{S/X\}$.

By coinduction, $G1\{I/X\}$ implements $M1$ with t:

$$G1\{I/X\} \overset{t}{\Rightarrow} P', G1\{S/X\} \overset{s}{\Rightarrow} Q', s \in M1, t \upharpoonright \overline{\mathcal{A}}(G1\{S/X\}) = s, P' \succeq_w R \succeq_w Q'.$$

By **Sum**, $G\{I/X\} \overset{t}{\Rightarrow} P'$ and $G\{S/X\} \overset{s}{\Rightarrow} Q'$.

$t \upharpoonright \overline{\mathcal{A}}(G\{S/X\}) = s$ unless $\overline{\mathcal{A}}(G2\{I/X\})$ contains actions in $\overline{\mathcal{Extr}}(G\{I/X\},$

$G\{S/X\})$.

Yet PEA assures that no $\bar{a}$ is in *both* $\overline{\mathcal{A}}(G2\{I/X\})$ and $\overline{\mathcal{E}xtr}(\,G\{I/X\}$.

$\therefore\ t\!\restriction\overline{\mathcal{A}}(G\{S/X\}) = s$.

$P' \succeq_w \mathcal{R}\succeq_w Q'$ is already known.

(The case where $G2\{I/X\}$ implements $M2$ is similar, but unneeded.)

$G \equiv G1|G2$.

When $G\{S/X\}$ has maxoctset M with respect to m then by Lemma 4-12,

$\quad\quad G1\{S/X\}$ has maxoctset $M1$ with respect to m_1

$\quad\quad G2\{S/X\}$ has maxoctset $M2$ with respect to m_2,

where $m =_{conf} m_1.m_2$

By coinduction, $G1\{I/X\}$ implements $M1$ with t_1:

$\quad G1\{I/X\} \overset{t_1}{\Rightarrow} P1',\ G1\{S/X\} \overset{s_1}{\Rightarrow} Q1',\ t_1\!\restriction\overline{\mathcal{A}}(G1\{S/X\}) = s_1 \in M1,\ P1' \succeq_w \mathcal{R}\succeq_w Q1'$.

and $G2\{I/X\}$ implements $M2$ with t_2:

$\quad G2\{I/X\} \overset{t_2}{\Rightarrow} P2',\ G2\{S/X\} \overset{s_2}{\Rightarrow} Q2',\ t_2\!\restriction\overline{\mathcal{A}}(G2\{S/X\}) = s_2 \in M2,\ P2' \succeq_w \mathcal{R}\succeq_w Q2'$.

Now $G\{I/X\} \overset{t_1}{\Rightarrow} \overset{t_2}{\Rightarrow} P1'|P2',\ G\{S/X\} \overset{s_1}{\Rightarrow} \overset{s_2}{\Rightarrow} Q1'|Q2',\ t_1 .t_2\!\restriction\overline{\mathcal{A}}(G\{S/X\}) = s_1 . s_2 \in M$.

$P1'|P2' \succeq_w \mathcal{R} \succeq_w Q1'|Q2'$ is shown by the composite H scheme.

$G \equiv G1\backslash\{c\}$.

Let $G\{S/X\} \equiv G1\{S/X\}\backslash\{c\}$ have maxoctset M with respect to m terminating at Q'.

Since all strings in a maxoctset are $=_{conf}$, $\backslash\{c\}$ blocks *every* string, or *none* of

them.

Hence Restriction preserves every maxoctset that it does not delete entirely.

$\therefore \bar{c} \notin m$, and M is a maxoctset of $G1\{S/X\}$.

By coinduction, $G1\{I/X\}$ implements M with t:

$\quad\quad G1\{I/X\} \overset{t}{\Rightarrow} P1' \succeq_w \mathcal{R}\succeq_w Q1',\ G1\{S/X\} \overset{s}{\Rightarrow} Q1'$, and $t\!\restriction\overline{\mathcal{A}}(G1\{S/X\}) = s \in M$.

To prove that $G1\{I/X\}\backslash\{c\} \overset{t}{\Rightarrow} P1'\backslash\{c\}$ one must be assured that $\bar{c} \notin t$.

Trial Hypothesis. Assume $\bar{c} \in t$.

Now $\bar{c} \notin s$ since s survived the Restriction.

$\therefore \bar{c} \in \overline{\mathcal{E}xtr}(G1\{I/X\}, G1\{S/X\})$.

B-14

COR requires that only *idle* extraneous outputs can be Restricted.

Lying along an implementing path t, the $\bar{c}$ is manisfestly *not* idle.

$\Rightarrow\Leftarrow$

Hence $G1\{I/X\}\backslash\{c\} \stackrel{t}{\Rightarrow} P1'\backslash\{c\}$ and $t\!\restriction\!\overline{\mathcal{A}}(G1\{S/X\}\backslash\{c\}) = s$.

$P1'\backslash\{c\} \succeq_w \mathcal{R} \succeq_w Q1'\backslash\{c\}$ is shown using the composite H scheme.

$G \equiv G1[f]$.

BR assures that f^{-1} exists and is bijective.

If $G1\{S/X\}[f]$ has maxoctset M then $G1\{S/X\}$ has maxocset $M[f^{-1}]$.

By coinduction, $G1\{I/X\}$ implements $M[f^{-1}]$ with t:

$\qquad G1\{I/X\} \stackrel{t}{\Rightarrow} P1' \succeq_w \mathcal{R} \succeq_w Q1'$, where $G1\{S/X\} \stackrel{s}{\Rightarrow} Q1'$ and

$\qquad t\!\restriction\!\overline{\mathcal{A}}(G1\{S/X\}) = s \in M[f^{-1}]$.

Hence,

$\qquad G1\{I/X\}[f] \stackrel{f(t)}{\Rightarrow} P1'[f]$ where $G1\{S/X\}[f] \stackrel{f(s)}{\Rightarrow} Q1'[f]$

$\qquad f(t)\!\restriction\!\overline{\mathcal{A}}(G1\{S/X\}[f]) = f(s) \in M$.

To show $P1'[f] \succeq_w \mathcal{R} \succeq_w Q1'[f]$ use the composite H scheme.

$G \equiv C$.

$G\{I/X\} \equiv G\{S/X\}$ and the satisfaction of LSIT' is trivial.

- *LII'*.

To show $\forall \gamma \in \mathcal{A}(G\{S/X\})$

Whenever $G\{I/X\} \stackrel{\gamma}{\rightarrow} P'$ and $G\{S/X\} \stackrel{\gamma}{\Rightarrow}$ then $G\{S/X\} \stackrel{\gamma}{\Rightarrow} Q'$ where $P' \succeq_w \mathcal{R} \succeq_w Q'$.

$G \equiv X$.

$G\{I/X\} \equiv I \stackrel{\text{def}}{=} E\{I/X\}$

When $G\{S/X\} \stackrel{\gamma}{\Rightarrow}$ the actions are inferred from **Con** where $S \stackrel{\text{def}}{=} F\{S/X\}$.

By a shorter inference $F\{S/X\} \stackrel{\gamma}{\Rightarrow}$.

Now $E\{S/X\} \succeq_w F\{S/X\}$ so LSAI requires that $E\{S/X\} \stackrel{\gamma}{\Rightarrow}$.

When $G\{I/X\} \equiv I \stackrel{\gamma}{\rightarrow} P'$ the action is inferred from **Con**, where $I \stackrel{\text{def}}{=} E\{I/X\}$.

By a shorter inference, $E\{I/X\} \stackrel{\gamma}{\rightarrow} P'$.

Since $E\{S/X\} \overset{\gamma}{\Rightarrow}$ then by coinduction, $E\{S/X\} \overset{\gamma}{\Rightarrow} Q'$ with $P' \succeq_w \mathcal{R} \succeq_w Q'$.

Now $E\{S/X\} \succeq_w F\{S/X\}$, and since $F\{S/X\} \overset{\gamma}{\Rightarrow}$ one may apply LII.

$F\{S/X\} \overset{\gamma}{\Rightarrow} R'$ where $Q' \succeq_w R'$.

So $G\{I/X\} \equiv I \overset{def}{=} E\{I/X\} \overset{\gamma}{\Rightarrow} P'$ and $G\{S/X\} \equiv S \overset{def}{=} F\{S/X\} \overset{\gamma}{\Rightarrow} R'$.

$P' \succeq_w \mathcal{R} \succeq_w Q' \succeq_w R'$, or more simply, $P' \succeq_w \mathcal{R} \succeq_w R'$.

$G \equiv \alpha.G1$.

α is an input.

Clearly $G\{I/X\} \overset{\alpha}{\rightarrow} G1\{I/X\}$, $G\{S/X\} \overset{\alpha}{\rightarrow} G1\{S/X\}$ and $G1\{S/X\} \, \mathcal{R} \, G1\{S/X\}$.

$G \equiv G1 + G2$.

Let $G\{I/X\} \overset{\gamma}{\rightarrow} P'$ and $G\{S/X\} \overset{\gamma}{\Rightarrow}$.

By a shorter inference on **Sum**, and w.l.o.g., $G1\{I/X\} \overset{\gamma}{\rightarrow} P'$.

The inference that $G1\{I/X\} \overset{\gamma}{\rightarrow}$ is independent of the instantiated agent variable.

$\therefore G1\{S/X\} \overset{\gamma}{\rightarrow}$ or, $G1\{S/X\} \overset{\gamma}{\Rightarrow}$.

By coinduction, $G1\{S/X\} \overset{\gamma}{\Rightarrow} Q'$ and $P' \succeq_w \mathcal{R} \succeq_w Q'$.

$G \equiv G1|G2$.

Let $G\{I/X\} \equiv G1\{I/X\}|G2\{I/X\} \overset{\gamma}{\rightarrow} P'$ and $G\{S/X\} \equiv G1\{S/X\}|G2\{S/X\} \overset{\gamma}{\Rightarrow}$.

By a shorter inference on **Com**, and w.l.o.g., $G1\{I/X\} \overset{\gamma}{\rightarrow} P''$.

Again, $G1\{S/X\} \overset{\gamma}{\Rightarrow}$ since the inference is independent of instantiated agent.

By coinduction $G1\{S/X\} \overset{\gamma}{\Rightarrow} Q''$ with $P'' \succeq_w \mathcal{R} \succeq_w Q''$.

Hence $G\{I/X\} \overset{\gamma}{\rightarrow} P'' \mid G2\{I/X\}$ and $G\{S/X\} \overset{\gamma}{\Rightarrow} Q'' \mid G2\{S/X\}$.

$P'' \mid G2\{I/X\} \succeq_w \mathcal{R} \succeq_w Q'' \mid G\{S/X\}$ is shown using the composite H scheme.

$G \equiv G1\backslash\{c\}$.

Let $G1\{I/X\}\backslash\{c\} \overset{\gamma}{\rightarrow} P'$ and $G1\{S/X\} \equiv G1\{S/X\}\backslash\{c\} \overset{\gamma}{\Rightarrow}$. $\gamma \neq c$.

By shorter inferences on **Res**, $G1\{I/X\} \overset{\gamma}{\rightarrow} P''$ and $G1\{S/X\} \overset{\gamma}{\Rightarrow}$ where $P' = P''\backslash\{c\}$.

By coinduction $G1\{S/X\} \overset{\gamma}{\Rightarrow} Q''$ with $P'' \succeq_w \mathcal{R} \succeq_w Q''$.

Hence $G\{I/X\} \overset{\gamma}{\rightarrow} P''\backslash\{c\}$ and $G\{S/X\} \overset{\gamma}{\Rightarrow} Q''\backslash\{c\}$.

$P''\backslash\{c\} \succeq_w \mathcal{R} \succeq_w Q''\backslash\{c\}$ is shown using the composite H scheme.

$G \equiv G1[f]$

$\quad$ *BR* assures that f^{-1} exists and is bijective.

$\quad$ Let $G1\{I/X\}[f] \xrightarrow{\gamma} P'[f]$ and $G1\{S/X\}[f] \xRightarrow{\gamma}$.

$\quad$ By shorter inferences, $G1\{I/X\} \xrightarrow{f^{-1}(\gamma)} P'$ and $G1\{S/X\} \xRightarrow{f^{-1}(\gamma)}$.

$\quad$ By coinduction, $G1\{S/X\} \xRightarrow{f^{-1}(\gamma)} Q'$ where $P' \succeq_w \mathcal{R} \succeq_w Q'$.

$\quad$ Hence, $G1\{S/X\}[f] \xRightarrow{\gamma} Q'[f]$

$\quad$ The proof that $P'[f] \succeq_w \mathcal{R} \succeq_w Q'[f]$ follows the composite H scheme.

$G \equiv C.$

$\quad$ $G\{I/X\} \equiv G\{S/X\}$ and the satisfaction of LII' is trivial.

- *LIOT'.*

$\quad$ To show $\forall \beta \in \overline{\mathcal{A}}(G\{I/X\}) \cup \{\tau\}$

$\quad$ If $G\{I/X\} \xrightarrow{\beta} P'$ then $G\{S/X\} \xRightarrow{\delta} Q'$ where $P' \succeq_w \mathcal{R} \succeq_w Q'$ and $\delta = \beta {\upharpoonright} \overline{\mathcal{A}}(G\{S/X\})$.

$G \equiv X.$

$\quad$ Let $G\{I/X\} \equiv I \xrightarrow{\beta} P'$.

$\quad$ $I \overset{def}{=} E\{I/X\}$ so, by a shorter inference, $E\{I/X\} \xrightarrow{\beta} P'$.

$\quad$ By coinduction, $E\{S/X\} \xRightarrow{\delta} Q'$ where $P' \succeq_w \mathcal{R} \succeq_w Q'$ and $\delta = \beta {\upharpoonright} \overline{\mathcal{A}}(G\{S/X\})$.

$\quad$ Apply $E \succeq_w F$. $E\{S/X\} \succeq_w F\{S/X\}$.

$\quad$ By LIOT $F\{S/X\} \xRightarrow{\delta} R'$ where $Q' \succeq_w R'$.

$\quad$ $S \overset{def}{=} F\{S/X\}$ so by **Con**, $G\{S/X\} \equiv S \xRightarrow{\delta} R'$.

$\quad$ $P' \succeq_w \mathcal{R} \succeq_w Q' \succeq_w R'$, so $P' \succeq_w \mathcal{R} \succeq_w R'$.

$G \equiv \alpha.G1.$

$\quad$ α is an output or τ.

$\quad$ $G\{I/X\} \xrightarrow{\alpha} G1\{I/X\}$, $G\{S/X\} \xRightarrow{\alpha} G1\{S/X\}$, $\alpha = \alpha {\upharpoonright} \overline{\mathcal{A}}(G\{S/X\})$, $G1\{I/X\} \mathcal{R} G1\{S/X\}$.

$G \equiv G1 + G2.$

$\quad$ Let $G1\{I/X\} + G2\{I/X\} \xrightarrow{\beta} P'$.

$\quad$ By a shorter inference on **Sum**, and w.l.o.g., $G1\{I/X\} \xrightarrow{\beta} P'$.

$\quad$ By coinduction, $G1\{S/X\} \xRightarrow{\delta} Q'$ where $P' \succeq_w \mathcal{R} \succeq_w Q'$ and $\delta = \beta {\upharpoonright} \overline{\mathcal{A}}(G1\{S/X\})$.

By **Sum**, $G\{S/X\} \equiv G1\{S/X\} + G2\{S/X\} \overset{\delta}{\Rightarrow} Q'$.

Since $\overline{\mathcal{A}}(G1\{S/X\}) \subseteq \overline{\mathcal{A}}(G\{S/X\})$ then $\beta \upharpoonright \overline{\mathcal{A}}(G\{S/X\}) = \delta$.

$P' \succeq_w \mathcal{R} \succeq_w Q'$ is already established.

$G \equiv G1|G2$.

    Let $G\{I/X\} \overset{\beta}{\rightarrow} P'$.

    By a shorter inference on **Com**, w.l.o.g., $G1\{I/X\} \overset{\beta}{\rightarrow} P''$ where $P' \equiv P''|G2\{I/X\}$.

    By coinduction, $G1\{S/X\} \overset{\delta}{\Rightarrow} Q''$ where $P'' \succeq_w \mathcal{R} \succeq_w Q''$ and $\delta = \beta \upharpoonright \overline{\mathcal{A}}(G1\{S/X\})$.

    By **Com**, $G\{S/X\} \overset{\delta}{\Rightarrow} Q''|G2\{S/X\}$.

    Since $\overline{\mathcal{A}}(G1\{S/X\}) \subseteq \overline{\mathcal{A}}(G\{S/X\})$ then $\beta \upharpoonright \overline{\mathcal{A}}(G\{S/X\}) = \delta$.

    $P''|G2\{I/X\} \succeq_w \mathcal{R} \succeq_w Q''|G2\{S/X\}$ is shown by the composite H scheme.

$G \equiv G1 \backslash \{c\}$.

    Let $G1\{I/X\}\backslash\{c\} \overset{\beta}{\rightarrow} P'$. $\beta \neq \{c\}$.

    By a shorter inference on **Res**, $G1\{I/X\} \overset{\beta}{\rightarrow} P''$ where $P''\backslash\{c\} = P'$.

    By coinduction, $G1\{S/X\} \overset{\delta}{\Rightarrow} Q''$ where $P'' \succeq_w \mathcal{R} \succeq_w Q''$ and $\delta = \beta \upharpoonright \overline{\mathcal{A}}(G1\{S/X\})$.

    By **Res,** $G1\{S/X\}\backslash\{c\} \overset{\delta}{\Rightarrow} Q' = Q''\backslash\{c\}$.

    $\beta \upharpoonright \overline{\mathcal{A}}(G1\{S/X\}\backslash\{c\}) = \delta$ since $\beta \neq \{c\}$.

    $P''\backslash\{c\} \succeq_w \mathcal{R} \succeq_w Q''\backslash\{c\}$ is shown by the composite H scheme.

$G \equiv G1[f]$.

    BR assures that f^{-1} exists and is bijective.

    Let $G1\{I/X\}[f] \overset{\beta}{\rightarrow} P'$.

    By a shorter inference on **Rel**, $G1\{I/X\} \overset{f^{-1}(\beta)}{\rightarrow} P''$ where $P' = P''[f]$.

    By coinduction,

        $G1\{S/X\} \overset{\delta}{\Rightarrow} Q''$ where $P'' \succeq_w \mathcal{R} \succeq_w Q''$ and $\delta = f^{-1}(\beta) \upharpoonright \overline{\mathcal{A}}(G1\{S/X\})$.

    By **Rel**, $G1\{S/X\}[f] \overset{f(\delta)}{\Rightarrow} Q''[f] = Q'$.

    $f(\delta) = f(f^{-1}(\beta) \upharpoonright \overline{\mathcal{A}}(G1\{S/X\})) = \beta \upharpoonright \overline{\mathcal{A}}(G1\{S/X\}[f]) = \beta \upharpoonright \overline{\mathcal{A}}(G\{S/X\})$.

    $P''[f] \succeq_w \mathrm{R} \succeq_w Q''[f]$ is shown by the composite H scheme.

$G \equiv C$.

$G\{I/X\} \equiv G\{S/X\}$ and the satisfaction of LII' is trivial.

◻

Appendix C

CCS Transition Rules (Milner, 1989)

Act $\dfrac{}{\alpha.E \xrightarrow{\alpha} E}$

Sum₁ $\dfrac{E \xrightarrow{\alpha} E'}{E + F \xrightarrow{\alpha} E'}$ **Sum₂** $\dfrac{E \xrightarrow{\alpha} E'}{F + E \xrightarrow{\alpha} E'}$

Com₁ $\dfrac{E \xrightarrow{\alpha} E'}{E \mid F \xrightarrow{\alpha} E' \mid F}$ **Com₂** $\dfrac{E \xrightarrow{\alpha} E'}{F \mid E \xrightarrow{\alpha} F \mid E'}$ **Com₃** $\dfrac{E \xrightarrow{\ell} E', F \xrightarrow{\bar{\ell}} E'}{E \mid F \xrightarrow{\tau} E' \mid F'}$

Res $\dfrac{E \xrightarrow{\alpha} E'}{E \backslash L \xrightarrow{\alpha} E' \backslash L}\ (\alpha, \bar{\alpha} \notin L)$

Rel $\dfrac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}$

Con $\dfrac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'}\ (A \overset{def}{=} P)$

$$\textbf{Rec} \quad \frac{E(\textit{fix}(X = E)) \xrightarrow{\alpha} E'}{\textit{fix}(X = E) \xrightarrow{\alpha} E'}$$

$\alpha \in \mathcal{A}\textit{ct}$, $\ell \in \mathcal{L}$, A and B are agents, E and F are agent expressions, and the restriction set $L \subseteq \mathcal{L}$.

These rules are implications with the upper transition(s) implying the lower. Side conditions apply for the rules **Res** and **Con**. The **Act** rule is universally inferred, having an empty premise (*truth*).

Appendix D

S, I and J Initial Models

```
entity S is
   port (    A, B, C, D : in bit;
         O0, O1, O2,…O9 : out bit );
end S;

architecture BEHAVIOR of S is
begin
   process (A,B,C,D)
   begin
      assert A&B&S&D < "1010";
      case A&B&S&D is
         when 0000 =>
            O0 <= 1;
            O1 <= 0;
            O2 <= 0;
            O3 <= 0;
            O4 <= 0;
            O5 <= 0;
            O6 <= 0;
            O7 <= 0;
            O8 <= 0;
            O9 <= 0;
         when 0001 =>
            O0 <= 0;
            O1 <= 1;
            O2 <= 0;
            O3 <= 0;
            O4 <= 0;
            O5 <= 0;
            O6 <= 0;
            O7 <= 0;
            O8 <= 0;
            O9 <= 0;
         when 0010 =>
            O0 <= 0;
            O1 <= 0;
            O2 <= 1;
            O3 <= 0;
            O4 <= 0;
            O5 <= 0;
            O6 <= 0;
            O7 <= 0;
            O8 <= 0;
            O9 <= 0;
         when 0011 =>
            O0 <= 0;
            O1 <= 0;
            O2 <= 0;
            O3 <= 1;
            O4 <= 0;
            O5 <= 0;
```

```
        O6 <= 0;
        O7 <= 0;
        O8 <= 0;
        O9 <= 0;
    when 0100 =>
        O0 <= 0;
        O1 <= 0;
        O2 <= 0;
        O3 <= 0;
        O4 <= 1;
        O5 <= 0;
        O6 <= 0;
        O7 <= 0;
        O8 <= 0;
        O9 <= 0;
    when 0101 =>
        O0 <= 0;
        O1 <= 0;
        O2 <= 0;
        O3 <= 0;
        O4 <= 0;
        O5 <= 1;
        O6 <= 0;
        O7 <= 0;
        O8 <= 0;
        O9 <= 0;
    when 0110 =>
        O0 <= 0;
        O1 <= 0;
        O2 <= 0;
        O3 <= 0;
        O4 <= 0;
        O5 <= 0;
        O6 <= 1;
        O7 <= 0;
        O8 <= 0;
        O9 <= 0;
    when 0111 =>
        O0 <= 0;
        O1 <= 0;
        O2 <= 0;
        O3 <= 0;
        O4 <= 0;
        O5 <= 0;
        O6 <= 0;
        O7 <= 1;
        O8 <= 0;
        O9 <= 0;

    when 1000 =>
        O0 <= 0;
        O1 <= 0;
        O2 <= 0;
        O3 <= 0;
        O4 <= 0;
        O5 <= 0;
        O6 <= 0;
        O7 <= 0;
        O8 <= 1;
        O9 <= 0;
```

```
        when 1001 =>
            O0 <= 0;
            O1 <= 0;
            O2 <= 0;
            O3 <= 0;
            O4 <= 0;
            O5 <= 0;
            O6 <= 0;
            O7 <= 0;
            O8 <= 0;
            O9 <= 1;
            end case;
        end process;
    end BEHAVIOR;
end behavior;
```

```
entity I is
    port (     A, B, C, D : in bit;
           O0, O1, O2,…O15 : out bit );
end S;

architecture BEHAVIOR of I is
begin
    process (A,B,C,D)
    begin
    case A&B&S&D is
           when 0000 =>
               O0 <= 1;
               O1 <= 0;
               O2 <= 0;
               O3 <= 0;
               O4 <= 0;
               O5 <= 0;
               O6 <= 0;
               O7 <= 0;
               O8 <= 0;
               O9 <= 0;
               O10 <= 0;
               O11 <= 0;
               O12 <= 0;
               O13 <= 0;
               O14 <= 0;
               O15 <= 0;
           when 0001 =>
               O0 <= 0;
               O1 <= 1;
               O2 <= 0;
               O3 <= 0;
               O4 <= 0;
               O5 <= 0;
               O6 <= 0;
               O7 <= 0;
               O8 <= 0;
               O9 <= 0;
               O10 <= 0;
               O11 <= 0;
               O12 <= 0;
               O13 <= 0;
               O14 <= 0;
               O15 <= 0;
           when 0010 =>
               O0 <= 0;
               O1 <= 0;
               O2 <= 1;
               O3 <= 0;
               O4 <= 0;
               O5 <= 0;
               O6 <= 0;
               O7 <= 0;
               O8 <= 0;
               O9 <= 0;
               O10 <= 0;
               O11 <= 0;
               O12 <= 0;
               O13 <= 0;
               O14 <= 0;
```

```
        O15 <= 0;
when 0011 =>
    O0 <= 0;
    O1 <= 0;
    O2 <= 0;
    O3 <= 1;
    O4 <= 0;
    O5 <= 0;
    O6 <= 0;
    O7 <= 0;
    O8 <= 0;
    O9 <= 0;
    O10 <= 0;
    O11 <= 0;
    O12 <= 0;
    O13 <= 0;
    O14 <= 0;
    O15 <= 0;
when 0100 =>
    O0 <= 0;
    O1 <= 0;
    O2 <= 0;
    O3 <= 0;
    O4 <= 1;
    O5 <= 0;
    O6 <= 0;
    O7 <= 0;
    O8 <= 0;
    O9 <= 0;
    O10 <= 0;
    O11 <= 0;
    O12 <= 0;
    O13 <= 0;
    O14 <= 0;
    O15 <= 0;
when 0101 =>
    O0 <= 0;
    O1 <= 0;
    O2 <= 0;
    O3 <= 0;
    O4 <= 0;
    O5 <= 1;
    O6 <= 0;
    O7 <= 0;
    O8 <= 0;
    O9 <= 0;
    O10 <= 0;
    O11 <= 0;
    O12 <= 0;
    O13 <= 0;
    O14 <= 0;
    O15 <= 0;
when 0110 =>
    O0 <= 0;
    O1 <= 0;
    O2 <= 0;
    O3 <= 0;
    O4 <= 0;
    O5 <= 0;
    O6 <= 1;
```

```
            O7  <= 0;
            O8  <= 0;
            O9  <= 0;
            O10 <= 0;
            O11 <= 0;
            O12 <= 0;
            O13 <= 0;
            O14 <= 0;
            O15 <= 0;
        when 0111 =>
            O0  <= 0;
            O1  <= 0;
            O2  <= 0;
            O3  <= 0;
            O4  <= 0;
            O5  <= 0;
            O6  <= 0;
            O7  <= 1;
            O8  <= 0;
            O9  <= 0;
            O10 <= 0;
            O11 <= 0;
            O12 <= 0;
            O13 <= 0;
            O14 <= 0;
            O15 <= 0;
        when 1000 =>
            O0  <= 0;
            O1  <= 0;
            O2  <= 0;
            O3  <= 0;
            O4  <= 0;
            O5  <= 0;
            O6  <= 0;
            O7  <= 0;
            O8  <= 1;
            O9  <= 0;
            O10 <= 0;
            O11 <= 0;
            O12 <= 0;
            O13 <= 0;
            O14 <= 0;
            O15 <= 0;
        when 1001 =>
            O0  <= 0;
            O1  <= 0;
            O2  <= 0;
            O3  <= 0;
            O4  <= 0;
            O5  <= 0;
            O6  <= 0;
            O7  <= 0;
            O8  <= 0;
            O9  <= 1;
            O10 <= 0;
            O11 <= 0;
            O12 <= 0;
            O13 <= 0;
            O14 <= 0;
            O15 <= 0;
```

```
when 1010 =>
    O0  <= 0;
    O1  <= 0;
    O2  <= 0;
    O3  <= 0;
    O4  <= 0;
    O5  <= 0;
    O6  <= 0;
    O7  <= 0;
    O8  <= 0;
    O9  <= 0;
    O10 <= 1;
    O11 <= 0;
    O12 <= 0;
    O13 <= 0;
    O14 <= 0;
    O15 <= 0;
when 1011 =>
    O0  <= 0;
    O1  <= 0;
    O2  <= 0;
    O3  <= 0;
    O4  <= 0;
    O5  <= 0;
    O6  <= 0;
    O7  <= 0;
    O8  <= 0;
    O9  <= 0;
    O10 <= 0;
    O11 <= 1;
    O12 <= 0;
    O13 <= 0;
    O14 <= 0;
    O15 <= 0;
when 1100 =>
    O0  <= 0;
    O1  <= 0;
    O2  <= 0;
    O3  <= 0;
    O4  <= 0;
    O5  <= 0;
    O6  <= 0;
    O7  <= 0;
    O8  <= 0;
    O9  <= 0;
    O10 <= 0;
    O11 <= 0;
    O12 <= 1;
    O13 <= 0;
    O14 <= 0;
    O15 <= 0;
when 1101 =>
    O0  <= 0;
    O1  <= 0;
    O2  <= 0;
    O3  <= 0;
    O4  <= 0;
    O5  <= 0;
    O6  <= 0;
    O7  <= 0;
```

```vhdl
                O8  <= 0;
                O9  <= 0;
                O10 <= 0;
                O11 <= 0;
                O12 <= 0;
                O13 <= 1;
                O14 <= 0;
                O15 <= 0;
            when 1110 =>
                O0  <= 0;
                O1  <= 0;
                O2  <= 0;
                O3  <= 0;
                O4  <= 0;
                O5  <= 0;
                O6  <= 0;
                O7  <= 0;
                O8  <= 0;
                O9  <= 0;
                O10 <= 0;
                O11 <= 0;
                O12 <= 0;
                O13 <= 0;
                O14 <= 1;
                O15 <= 0;
            when 1111 =>
                O0  <= 0;
                O1  <= 0;
                O2  <= 0;
                O3  <= 0;
                O4  <= 0;
                O5  <= 0;
                O6  <= 0;
                O7  <= 0;
                O8  <= 0;
                O9  <= 0;
                O10 <= 0;
                O11 <= 0;
                O12 <= 0;
                O13 <= 0;
                O14 <= 0;
                O15 <= 1;
            end case;
        end process;
end BEHAVIOR;
entity J is
   port (    A, B, C, D : in bit;
         O0, O1, O2,…O15 : out bit );
end S;

architecture BEHAVIOR of J is
   constant DELAY0,  DELAY1,  DELAY2,  DELAY3,
            DELAY4,  DELAY5,  DELAY6,  DELAY7,
            DELAY8,  DELAY9,  DELAY10, DELAY11,
            DELAY12, DELAY13, DELAY14, DELAY15
                                         :  time;
begin
   process (A,B,C,D)
   begin
```

```
case A&B&S&D is
    when 0000 =>
        O0 <= 1 after DELAY0;
        O1 <= 0 after DELAY1;
        O2 <= 0 after DELAY2;
        O3 <= 0 after DELAY3;
        O4 <= 0 after DELAY4;
        O5 <= 0 after DELAY5;
        O6 <= 0 after DELAY6;
        O7 <= 0 after DELAY7;
        O8 <= 0 after DELAY8;
        O9 <= 0 after DELAY9;
        O10 <= 0 after DELAY10;
        O11 <= 0 after DELAY11;
        O12 <= 0 after DELAY12;
        O13 <= 0 after DELAY13;
        O14 <= 0 after DELAY14;
        O15 <= 0 after DELAY15;
    when 0001 =>
        O0 <= 0 after DELAY0;
        O1 <= 1 after DELAY1;
        O2 <= 0 after DELAY2;
        O3 <= 0 after DELAY3;
        O4 <= 0 after DELAY4;
        O5 <= 0 after DELAY5;
        O6 <= 0 after DELAY6;
        O7 <= 0 after DELAY7;
        O8 <= 0 after DELAY8;
        O9 <= 0 after DELAY9;
        O10 <= 0 after DELAY10;
        O11 <= 0 after DELAY11;
        O12 <= 0 after DELAY12;
        O13 <= 0 after DELAY13;
        O14 <= 0 after DELAY14;
        O15 <= 0 after DELAY15;

    when 0010 =>
        O0 <= 0 after DELAY0;
        O1 <= 0 after DELAY1;
        O2 <= 1 after DELAY2;
        O3 <= 0 after DELAY3;
        O4 <= 0 after DELAY4;
        O5 <= 0 after DELAY5;
        O6 <= 0 after DELAY6;
        O7 <= 0 after DELAY7;
        O8 <= 0 after DELAY8;
        O9 <= 0 after DELAY9;
        O10 <= 0 after DELAY10;
        O11 <= 0 after DELAY11;
        O12 <= 0 after DELAY12;
        O13 <= 0 after DELAY13;
        O14 <= 0 after DELAY14;
        O15 <= 0 after DELAY15;
    when 0011 =>
        O0 <= 0 after DELAY0;
        O1 <= 0 after DELAY1;
        O2 <= 0 after DELAY2;
        O3 <= 1 after DELAY3;
        O4 <= 0 after DELAY4;
        O5 <= 0 after DELAY5;
```

```
        O6 <= 0 after DELAY6;
        O7 <= 0 after DELAY7;
        O8 <= 0 after DELAY8;
        O9 <= 0 after DELAY9;
        O10 <= 0 after DELAY10;
        O11 <= 0 after DELAY11;
        O12 <= 0 after DELAY12;
        O13 <= 0 after DELAY13;
        O14 <= 0 after DELAY14;
        O15 <= 0 after DELAY15;
    when 0100 =>
        O0 <= 0 after DELAY0;
        O1 <= 0 after DELAY1;
        O2 <= 0 after DELAY2;
        O3 <= 0 after DELAY3;
        O4 <= 1 after DELAY4;
        O5 <= 0 after DELAY5;
        O6 <= 0 after DELAY6;
        O7 <= 0 after DELAY7;
        O8 <= 0 after DELAY8;
        O9 <= 0 after DELAY9;
        O10 <= 0 after DELAY10;
        O11 <= 0 after DELAY11;
        O12 <= 0 after DELAY12;
        O13 <= 0 after DELAY13;
        O14 <= 0 after DELAY14;
        O15 <= 0 after DELAY15;
    when 0101 =>
        O0 <= 0 after DELAY0;
        O1 <= 0 after DELAY1;
        O2 <= 0 after DELAY2;
        O3 <= 0 after DELAY3;
        O4 <= 0 after DELAY4;
        O5 <= 1 after DELAY5;
        O6 <= 0 after DELAY6;
        O7 <= 0 after DELAY7;
        O8 <= 0 after DELAY8;
        O9 <= 0 after DELAY9;
        O10 <= 0 after DELAY10;
        O11 <= 0 after DELAY11;
        O12 <= 0 after DELAY12;
        O13 <= 0 after DELAY13;
        O14 <= 0 after DELAY14;
        O15 <= 0 after DELAY15;
    when 0110 =>
        O0 <= 0 after DELAY0;
        O1 <= 0 after DELAY1;
        O2 <= 0 after DELAY2;
        O3 <= 0 after DELAY3;
        O4 <= 0 after DELAY4;
        O5 <= 0 after DELAY5;
        O6 <= 1 after DELAY6;
        O7 <= 0 after DELAY7;
        O8 <= 0 after DELAY8;
        O9 <= 0 after DELAY9;
        O10 <= 0 after DELAY10;
        O11 <= 0 after DELAY11;
        O12 <= 0 after DELAY12;
        O13 <= 0 after DELAY13;
        O14 <= 0 after DELAY14;
```

```
         O15 <= 0 after DELAY15;
      when 0111 =>
         O0 <= 9 after DELAY0;
         O1 <= 0 after DELAY1;
         O2 <= 0 after DELAY2;
         O3 <= 0 after DELAY3;
         O4 <= 0 after DELAY4;
         O5 <= 0 after DELAY5;
         O6 <= 0 after DELAY6;
         O7 <= 1 after DELAY7;
         O8 <= 0 after DELAY8;
         O9 <= 0 after DELAY9;
         O10 <= 0 after DELAY10;
         O11 <= 0 after DELAY11;
         O12 <= 0 after DELAY12;
         O13 <= 0 after DELAY13;
         O14 <= 0 after DELAY14;
         O15 <= 0 after DELAY15;
      when 1000 =>
         O0 <= 0 after DELAY0;
         O1 <= 0 after DELAY1;
         O2 <= 0 after DELAY2;
         O3 <= 0 after DELAY3;
         O4 <= 0 after DELAY4;
         O5 <= 0 after DELAY5;
         O6 <= 0 after DELAY6;
         O7 <= 0 after DELAY7;
         O8 <= 1 after DELAY8;
         O9 <= 0 after DELAY9;
         O10 <= 0 after DELAY10;
         O11 <= 0 after DELAY11;
         O12 <= 0 after DELAY12;
         O13 <= 0 after DELAY13;
         O14 <= 0 after DELAY14;
         O15 <= 0 after DELAY15;
      when 1001 =>
         O0 <= 0 after DELAY0;
         O1 <= 0 after DELAY1;
         O2 <= 0 after DELAY2;
         O3 <= 0 after DELAY3;
         O4 <= 0 after DELAY4;
         O5 <= 0 after DELAY5;
         O6 <= 0 after DELAY6;
         O7 <= 0 after DELAY7;
         O8 <= 0 after DELAY8;
         O9 <= 1 after DELAY9;
         O10 <= 0 after DELAY10;
         O11 <= 0 after DELAY11;
         O12 <= 0 after DELAY12;
         O13 <= 0 after DELAY13;
         O14 <= 0 after DELAY14;
         O15 <= 0 after DELAY15;
      when 1010 =>
         O0 <= 0 after DELAY0;
         O1 <= 0 after DELAY1;
         O2 <= 0 after DELAY2;
         O3 <= 0 after DELAY3;
         O4 <= 0 after DELAY4;
         O5 <= 0 after DELAY5;
         O6 <= 0 after DELAY6;
```

```
            O7 <= 0 after DELAY7;
            O8 <= 0 after DELAY8;
            O9 <= 0 after DELAY9;
            O10 <= 1 after DELAY10;
            O11 <= 0 after DELAY11;
            O12 <= 0 after DELAY12;
            O13 <= 0 after DELAY13;
            O14 <= 0 after DELAY14;
            O15 <= 0 after DELAY15;
        when 1011 =>
            O0 <= 0 after DELAY0;
            O1 <= 0 after DELAY1;
            O2 <= 0 after DELAY2;
            O3 <= 0 after DELAY3;
            O4 <= 0 after DELAY4;
            O5 <= 0 after DELAY5;
            O6 <= 0 after DELAY6;
            O7 <= 0 after DELAY7;
            O8 <= 0 after DELAY8;
            O9 <= 0 after DELAY9;
            O10 <= 0 after DELAY10;
            O11 <= 1 after DELAY11;
            O12 <= 0 after DELAY12;
            O13 <= 0 after DELAY13;
            O14 <= 0 after DELAY14;
            O15 <= 0 after DELAY15;
        when 1100 =>
            O0 <= 0 after DELAY0;
            O1 <= 0 after DELAY1;
            O2 <= 0 after DELAY2;
            O3 <= 0 after DELAY3;
            O4 <= 0 after DELAY4;
            O5 <= 0 after DELAY5;
            O6 <= 0 after DELAY6;
            O7 <= 0 after DELAY7;
            O8 <= 0 after DELAY8;
            O9 <= 0 after DELAY9;
            O10 <= 0 after DELAY10;
            O11 <= 0 after DELAY11;
            O12 <= 1 after DELAY12;
            O13 <= 0 after DELAY13;
            O14 <= 0 after DELAY14;
            O15 <= 0 after DELAY15;
        when 1101 =>
            O0 <= 0 after DELAY0;
            O1 <= 0 after DELAY1;
            O2 <= 0 after DELAY2;
            O3 <= 0 after DELAY3;
            O4 <= 0 after DELAY4;
            O5 <= 0 after DELAY5;
            O6 <= 0 after DELAY6;
            O7 <= 0 after DELAY7;
            O8 <= 0 after DELAY8;
            O9 <= 0 after DELAY9;
            O10 <= 0 after DELAY10;
            O11 <= 0 after DELAY11;
            O12 <= 0 after DELAY12;
            O13 <= 1 after DELAY13;
            O14 <= 0 after DELAY14;
            O15 <= 0 after DELAY15;
```

```
        when 1110 =>
           O0 <= 0 after DELAY0;
           O1 <= 0 after DELAY1;
           O2 <= 0 after DELAY2;
           O3 <= 0 after DELAY3;
           O4 <= 0 after DELAY4;
           O5 <= 0 after DELAY5;
           O6 <= 0 after DELAY6;
           O7 <= 0 after DELAY7;
           O8 <= 0 after DELAY8;
           O9 <= 0 after DELAY9;
           O10 <= 0 after DELAY10;
           O11 <= 0 after DELAY11;
           O12 <= 0 after DELAY12;
           O13 <= 0 after DELAY13;
           O14 <= 1 after DELAY14;
           O15 <= 0 after DELAY15;
        when 1111 =>
           O0 <= 0 after DELAY0;
           O1 <= 0 after DELAY1;
           O2 <= 0 after DELAY2;
           O3 <= 0 after DELAY3;
           O4 <= 0 after DELAY4;
           O5 <= 0 after DELAY5;
           O6 <= 0 after DELAY6;
           O7 <= 0 after DELAY7;
           O8 <= 0 after DELAY8;
           O9 <= 0 after DELAY9;
           O10 <= 0 after DELAY10;
           O11 <= 0 after DELAY11;
           O12 <= 0 after DELAY12;
           O13 <= 0 after DELAY13;
           O14 <= 0 after DELAY14;
           O15 <= 1 after DELAY15;
         end case;
       end process;
   end BEHAVIOR;
```

Appendix E

S,I and J Target Modesl

$S\_Behavior\_Default\_0 \overset{def}{=}$

      s_behavior_default_a.('s_behavior_default_o0| 's_behavior_default_o1).S_Behavior_Default_1

      + s_behavior_default_b.('s_behavior_default_o0| s_behavior_default_o2).S_Behavior_Default_2

      + s_behavior_default_c.('s_behavior_default_o0| 's_behavior_default_o4).S_Behavior_Default_4

      + s_behavior_default_d.('s_behavior_default_o0| 's_behavior_default_o8).S_Behavior_Default_8

$S\_Behavior\_Default\_1 \overset{def}{=}$

      s_behavior_default_a.('s_behavior_default_o1 | 's_behavior_default_o0).S_Behavior_Default_0

      + s_behavior_default_b.('s_behavior_default_o1 | 's_behavior_default_o3).S_Behavior_Default_3

      + s_behavior_default_c.('s_behavior_default_o1 | 's_behavior_default_o5).S_Behavior_Default_5

      + s_behavior_default_d.('s_behavior_default_o1 | 's_behavior_default_o9).S_Behavior_Default_9

$S\_Behavior\_Default\_2 \overset{def}{=}$

      s_behavior_default_a.('s_behavior_default_o2 | 's_behavior_default_o3).S_Behavior_Default_3

      + s_behavior_default_b.('s_behavior_default_o2 | 's_behavior_default_o0).S_Behavior_Default_0

      + s_behavior_default_c.('s_behavior_default_o2 | 's_behavior_default_o6).S_Behavior_Default_6

$S\_Behavior\_Default\_3 \overset{def}{=}$

      s_behavior_default_a.('s_behavior_default_o3 | 's_behavior_default_o2).S_Behavior_Default_2

      + s_behavior_default_b.('s_behavior_default_o3 | 's_behavior_default_o1).S_Behavior_Default_1

      + s_behavior_default_c.('s_behavior_default_o3 | 's_behavior_default_o7).S_Behavior_Default_7

$S\_Behavior\_Default\_4 \overset{def}{=}$

      s_behavior_default_a.('s_behavior_default_o4 | 's_behavior_default_o5).S_Behavior_Default_5

      + s_behavior_default_b.('s_behavior_default_o4 | 's_behavior_default_o6).S_Behavior_Default_6

      + s_behavior_default_c.('s_behavior_default_o4 | $\bar{o}_0$).S_Behavior_Default_0

$S\_Behavior\_Default\_5 \stackrel{def}{=}$

  $s\_behavior\_default\_a.(\ 's\_behavior\_default\_o5 \mid 's\_behavior\_default\_o4).S\_Behavior\_Default\_4$

  $+\ s\_behavior\_default\_b.(\ 's\_behavior\_default\_o5 \mid 's\_behavior\_default\_o7).S\_Behavior\_Default\_7$

  $+\ s\_behavior\_default\_c.(\ 's\_behavior\_default\_o5 \mid 's\_behavior\_default\_o1).S\_Behavior\_Default\_1$

$S\_Behavior\_Default\_6 \stackrel{def}{=}$

  $s\_behavior\_default\_a.(\ 's\_behavior\_default\_o6 \mid 's\_behavior\_default\_o7).S\_Behavior\_Default\_7$

  $+\ s\_behavior\_default\_b.(\ 's\_behavior\_default\_o6 \mid 's\_behavior\_default\_o4).S\_Behavior\_Default\_4$

  $+\ s\_behavior\_default\_c.(\ 's\_behavior\_default\_o6 \mid 's\_behavior\_default\_o2).S\_Behavior\_Default\_2$

$S\_Behavior\_Default\_7 \stackrel{def}{=}$

  $s\_behavior\_default\_a.(\ 's\_behavior\_default\_o7 \mid 's\_behavior\_default\_o6).S\_Behavior\_Default\_6$

  $+\ s\_behavior\_default\_b.(\ 's\_behavior\_default\_o7 \mid 's\_behavior\_default\_o5).S\_Behavior\_Default\_5$

  $+\ s\_behavior\_default\_c.(\ 's\_behavior\_default\_o7 \mid 's\_behavior\_default\_o3).S\_Behavior\_Default\_3$

$S\_Behavior\_Default\_8 \stackrel{def}{=}$

  $s\_behavior\_default\_a.(\ 's\_behavior\_default\_o8 \mid 's\_behavior\_default\_o9).S\_Behavior\_Default\_9$

  $+\ s\_behavior\_default\_d.(\ 's\_behavior\_default\_o8 \mid 's\_behavior\_default\_o0).S\_Behavior\_Default\_0$

$S\_Behavior\_Default\_9 \stackrel{def}{=}$

  $s\_behavior\_default\_a.(\ 's\_behavior\_default\_o9 \mid 's\_behavior\_default\_o8).S\_Behavior\_Default\_8$

  $+\ s\_behavior\_default\_d.(\ 's\_behavior\_default\_o9 \mid 's\_behavior\_default\_o1).S\_Behavior\_Default\_1$

$I\_Behavior\_Default\_0 \overset{def}{=}$

        $i\_behavior\_default\_a.(\ 'i\_behavior\_default\_o0|\ 'i\_behavior\_default\_o1).I\_Behavior\_Default\_1$

        $+\ i\_behavior\_default\_b.(\ 'i\_behavior\_default\_o0|\ i\_behavior\_default\_o2).I\_Behavior\_Default\_2$

        $+\ i\_behavior\_default\_c.(\ 'i\_behavior\_default\_o0|\ 'i\_behavior\_default\_o4).I\_Behavior\_Default\_4$

        $+\ i\_behavior\_default\_d.('i\_behavior\_default\_o0|\ 'i\_behavior\_default\_o8).I\_Behavior\_Default\_8$

$I\_Behavior\_Default\_1 \overset{def}{=}$

        $i\_behavior\_default\_a.(\ 'i\_behavior\_default\_o1\ |\ 'i\_behavior\_default\_o0).I\_Behavior\_Default\_0$

        $+\ i\_behavior\_default\_b.(\ 'i\_behavior\_default\_o1\ |\ 'i\_behavior\_default\_o3).I\_Behavior\_Default\_3$

        $+\ i\_behavior\_default\_c.(\ 'i\_behavior\_default\_o1\ |\ 'i\_behavior\_default\_o5).I\_Behavior\_Default\_5$

        $+\ i\_behavior\_default\_d.(\ 'i\_behavior\_default\_o1\ |\ 'i\_behavior\_default\_o9).I\_Behavior\_Default\_9$

$I\_Behavior\_Default\_2 \overset{def}{=}$

        $i\_behavior\_default\_a.(\ 'i\_behavior\_default\_o2\ |\ 'i\_behavior\_default\_o3).I\_Behavior\_Default\_3$

        $+\ i\_behavior\_default\_b.(\ 'i\_behavior\_default\_o2\ |\ 'i\_behavior\_default\_o0).I\_Behavior\_Default\_0$

        $+\ i\_behavior\_default\_c.(\ 'i\_behavior\_default\_o2\ |\ 'i\_behavior\_default\_o6).I\_Behavior\_Default\_6$

        $+\ i\_behavior\_default\_d.(\ 'i\_behavior\_default\_o2\ |\ 'i\_behavior\_default\_o10).I\_Behavior\_Default\_10$

$I\_Behavior\_Default\_3 \overset{def}{=}$

        $i\_behavior\_default\_a.(\ 'i\_behavior\_default\_o3\ |\ 'i\_behavior\_default\_o2).I\_Behavior\_Default\_2$

        $+\ i\_behavior\_default\_b.(\ 'i\_behavior\_default\_o3\ |\ 'i\_behavior\_default\_o1).I\_Behavior\_Default\_1$

        $+\ i\_behavior\_default\_c.(\ 'i\_behavior\_default\_o3\ |\ 'i\_behavior\_default\_o7).I\_Behavior\_Default\_7$

        $+\ i\_behavior\_default\_d.(\ 'i\_behavior\_default\_o3\ |\ 'i\_behavior\_default\_o11).I\_Behavior\_Default\_11$

$I\_Behavior\_Default\_4 \overset{def}{=}$

        $i\_behavior\_default\_a.(\ 'i\_behavior\_default\_o4\ |\ 'i\_behavior\_default\_o5).I\_Behavior\_Default\_5$

        $+\ i\_behavior\_default\_b.(\ 'i\_behavior\_default\_o4\ |\ 'i\_behavior\_default\_o6).I\_Behavior\_Default\_6$

        $+\ i\_behavior\_default\_c.(\ 'i\_behavior\_default\_o4\ |\ \bar{o}_0).I\_Behavior\_Default\_0$

        $+\ i\_behavior\_default\_d.(\ 'i\_behavior\_default\_o4\ |\ 'i\_behavior\_default\_o12).I\_Behavior\_Default\_12$

$I\_Behavior\_Default\_5 \overset{def}{=}$

        $i\_behavior\_default\_a.(\ 'i\_behavior\_default\_o5\ |\ 'i\_behavior\_default\_o4).I\_Behavior\_Default\_4$

$+ i\_behavior\_default\_b.('i\_behavior\_default\_o5 | 'i\_behavior\_default\_o7).I\_Behavior\_Default\_7$

$+ i\_behavior\_default\_c.('i\_behavior\_default\_o5 | 'i\_behavior\_default\_o1).I\_Behavior\_Default\_1$

$+ i\_behavior\_default\_d.('i\_behavior\_default\_o5 | 'i\_behavior\_default\_o13).I\_Behavior\_Default\_13$

$I\_Behavior\_Default\_6 \stackrel{def}{=}$

$i\_behavior\_default\_a.('i\_behavior\_default\_o6 | 'i\_behavior\_default\_o7).I\_Behavior\_Default\_7$

$+ i\_behavior\_default\_b.('i\_behavior\_default\_o6 | 'i\_behavior\_default\_o4).I\_Behavior\_Default\_4$

$+ i\_behavior\_default\_c.('i\_behavior\_default\_o6 | 'i\_behavior\_default\_o2).I\_Behavior\_Default\_2$

$+ i\_behavior\_default\_d.('i\_behavior\_default\_o6 | 'i\_behavior\_default\_o14).I\_Behavior\_Default\_14$

$I\_Behavior\_Default\_7 \stackrel{def}{=}$

$i\_behavior\_default\_a.('i\_behavior\_default\_o7 | 'i\_behavior\_default\_o6).I\_Behavior\_Default\_6$

$+ i\_behavior\_default\_b.('i\_behavior\_default\_o7 | 'i\_behavior\_default\_o5).I\_Behavior\_Default\_5$

$+ i\_behavior\_default\_c.('i\_behavior\_default\_o7 | 'i\_behavior\_default\_o3).I\_Behavior\_Default\_3$

$+ i\_behavior\_default\_d.('i\_behavior\_default\_o7 | 'i\_behavior\_default\_o15).I\_Behavior\_Default\_15$

$I\_Behavior\_Default\_8 \stackrel{def}{=}$

$i\_behavior\_default\_a.('i\_behavior\_default\_o8 | 'i\_behavior\_default\_o9).I\_Behavior\_Default\_9$

$+ i\_behavior\_default\_b.('i\_behavior\_default\_o8 | 'i\_behavior\_default\_o10).I\_Behavior\_Default\_10$

$+ i\_behavior\_default\_c.('i\_behavior\_default\_o8 | 'i\_behavior\_default\_o123).I\_Behavior\_Default\_12$

$+ i\_behavior\_default\_d.('i\_behavior\_default\_o8 | 'i\_behavior\_default\_o0).I\_Behavior\_Default\_0$

$I\_Behavior\_Default\_9 \stackrel{def}{=}$

$i\_behavior\_default\_a.('i\_behavior\_default\_o9 | 'i\_behavior\_default\_o8).I\_Behavior\_Default\_8$

$+ i\_behavior\_default\_b.('i\_behavior\_default\_o9 | 'i\_behavior\_default\_o11).I\_Behavior\_Default\_11$

$+ i\_behavior\_default\_c.('i\_behavior\_default\_o9 | 'i\_behavior\_default\_o13).I\_Behavior\_Default\_13$

$+ i\_behavior\_default\_d.('i\_behavior\_default\_o9 | 'i\_behavior\_default\_o1).I\_Behavior\_Default\_1$

$I\_Behavior\_Default\_10 \stackrel{def}{=}$

$i\_behavior\_default\_a.('i\_behavior\_default\_o10 | 'i\_behavior\_default\_o11).I\_Behavior\_Default\_11$

$+ i\_behavior\_default\_b.('i\_behavior\_default\_o10 | 'i\_behavior\_default\_o8).I\_Behavior\_Default\_8$

$+ i\_behavior\_default\_c.('i\_behavior\_default\_o10 | 'i\_behavior\_default\_o14).I\_Behavior\_Default\_14$

$+ i\_behavior\_default\_d.(\, 'i\_behavior\_default\_o10 \, | \, 'i\_behavior\_default\_o2).I\_Behavior\_Default\_2$

$I\_Behavior\_Default\_11 \stackrel{def}{=}$

$\quad i\_behavior\_default\_a.(\, 'i\_behavior\_default\_o11 \, | \, 'i\_behavior\_default\_o10).I\_Behavior\_Default\_10$

$\quad + i\_behavior\_default\_b.(\, 'i\_behavior\_default\_o11 \, | \, 'i\_behavior\_default\_o9).I\_Behavior\_Default\_9$

$\quad + i\_behavior\_default\_c.(\, 'i\_behavior\_default\_o11 \, | \, 'i\_behavior\_default\_o15).I\_Behavior\_Default\_15$

$\quad + i\_behavior\_default\_d.(\, 'i\_behavior\_default\_o11 \, | \, 'i\_behavior\_default\_o7).I\_Behavior\_Default\_7$

$I\_Behavior\_Default\_12 \stackrel{def}{=}$

$\quad i\_behavior\_default\_a.(\, 'i\_behavior\_default\_o12 \, | \, 'i\_behavior\_default\_o13).I\_Behavior\_Default\_13$

$\quad + i\_behavior\_default\_b.(\, 'i\_behavior\_default\_o12 \, | \, 'i\_behavior\_default\_o14).I\_Behavior\_Default\_14$

$\quad + i\_behavior\_default\_c.(\, 'i\_behavior\_default\_o12 \, | \, 'i\_behavior\_default\_o8).I\_Behavior\_Default\_8$

$\quad + i\_behavior\_default\_d.(\, 'i\_behavior\_default\_o12 \, | \, 'i\_behavior\_default\_o4).I\_Behavior\_Default\_4$

$I\_Behavior\_Default\_13 \stackrel{def}{=}$

$\quad i\_behavior\_default\_a.(\, 'i\_behavior\_default\_o13 \, | \, 'i\_behavior\_default\_o12).I\_Behavior\_Default\_12$

$\quad + i\_behavior\_default\_b.(\, 'i\_behavior\_default\_o13 \, | \, 'i\_behavior\_default\_o15).I\_Behavior\_Default\_15$

$\quad + i\_behavior\_default\_c.(\, 'i\_behavior\_default\_o13 \, | \, 'i\_behavior\_default\_o19).I\_Behavior\_Default\_19$

$\quad + i\_behavior\_default\_d.(\, 'i\_behavior\_default\_o13 \, | \, 'i\_behavior\_default\_o5).I\_Behavior\_Default\_5$

$I\_Behavior\_Default\_14 \stackrel{def}{=}$

$\quad i\_behavior\_default\_a.(\, 'i\_behavior\_default\_o14 \, | \, 'i\_behavior\_default\_o15).I\_Behavior\_Default\_15$

$\quad + i\_behavior\_default\_b.(\, 'i\_behavior\_default\_o14 \, | \, 'i\_behavior\_default\_o12).I\_Behavior\_Default\_12$

$\quad + i\_behavior\_default\_c.(\, 'i\_behavior\_default\_o14 \, | \, 'i\_behavior\_default\_o10).I\_Behavior\_Default\_10$

$\quad + i\_behavior\_default\_d.(\, 'i\_behavior\_default\_o14 \, | \, 'i\_behavior\_default\_o6).I\_Behavior\_Default\_6$

$I\_Behavior\_Default\_15 \stackrel{def}{=}$

$\quad i\_behavior\_default\_a.(\, 'i\_behavior\_default\_o15 \, | \, 'i\_behavior\_default\_o14).I\_Behavior\_Default\_14$

$\quad + i\_behavior\_default\_b.(\, 'i\_behavior\_default\_o15 \, | \, 'i\_behavior\_default\_o13).I\_Behavior\_Default\_13$

$\quad + i\_behavior\_default\_c.(\, 'i\_behavior\_default\_o15 \, | \, 'i\_behavior\_default\_o11).I\_Behavior\_Default\_11$

$\quad + i\_behavior\_default\_d.(\, 'i\_behavior\_default\_o15 \, | \, 'i\_behavior\_default\_o7).I\_Behavior\_Default\_7$

$J\_Behavior\_Default\_0 \stackrel{def}{=}$

> $j\_behavior\_default\_a. 'j\_behavior\_default\_o0. 'j\_behavior\_default\_o1.J\_Behavior\_Default\_1$
>
> $+ j\_behavior\_default\_b. 'j\_behavior\_default\_o0. j\_behavior\_default\_o2.J\_Behavior\_Default\_2$
>
> $+ j\_behavior\_default\_c. 'j\_behavior\_default\_o0. 'j\_behavior\_default\_o4.J\_Behavior\_Default\_4$
>
> $+ j\_behavior\_default\_d.'j\_behavior\_default\_o0. 'j\_behavior\_default\_o8.J\_Behavior\_Default\_8$

$J\_Behavior\_Default\_1 \stackrel{def}{=}$

> $j\_behavior\_default\_a. 'j\_behavior\_default\_o0 . 'j\_behavior\_default\_o1.J\_Behavior\_Default\_0$
>
> $+ j\_behavior\_default\_b. 'j\_behavior\_default\_o1 . 'j\_behavior\_default\_o3.J\_Behavior\_Default\_3$
>
> $+ j\_behavior\_default\_c. 'j\_behavior\_default\_o1 . 'j\_behavior\_default\_o5.J\_Behavior\_Default\_5$
>
> $+ j\_behavior\_default\_d. 'j\_behavior\_default\_o1 . 'j\_behavior\_default\_o9.J\_Behavior\_Default\_9$

$J\_Behavior\_Default\_2 \stackrel{def}{=}$

> $j\_behavior\_default\_a. 'j\_behavior\_default\_o2 . 'j\_behavior\_default\_o3.J\_Behavior\_Default\_3$
>
> $+ j\_behavior\_default\_b. 'j\_behavior\_default\_o0 . 'j\_behavior\_default\_o2.J\_Behavior\_Default\_0$
>
> $+ j\_behavior\_default\_c. 'j\_behavior\_default\_o2 . 'j\_behavior\_default\_o6.J\_Behavior\_Default\_6$
>
> $+ j\_behavior\_default\_d. 'j\_behavior\_default\_o2 . 'j\_behavior\_default\_o10.J\_Behavior\_Default\_10$

$J\_Behavior\_Default\_3 \stackrel{def}{=}$

> $j\_behavior\_default\_a. 'j\_behavior\_default\_o2 . 'j\_behavior\_default\_o3.J\_Behavior\_Default\_2$
>
> $+ j\_behavior\_default\_b. 'j\_behavior\_default\_o1 . 'j\_behavior\_default\_o3.J\_Behavior\_Default\_1$
>
> $+ j\_behavior\_default\_c. 'j\_behavior\_default\_o3 . 'j\_behavior\_default\_o7.J\_Behavior\_Default\_7$
>
> $+ j\_behavior\_default\_d. 'j\_behavior\_default\_o3 . 'j\_behavior\_default\_o11.J\_Behavior\_Default\_11$

$J\_Behavior\_Default\_4 \stackrel{def}{=}$

> $j\_behavior\_default\_a. 'j\_behavior\_default\_o4 . 'j\_behavior\_default\_o5.J\_Behavior\_Default\_5$
>
> $+ j\_behavior\_default\_b. 'j\_behavior\_default\_o4 . 'j\_behavior\_default\_o6.J\_Behavior\_Default\_6$
>
> $+ j\_behavior\_default\_c. 'j\_behavior\_default\_o0 .'j\_behavior\_default\_4.J\_Behavior\_Default\_0$
>
> $+ j\_behavior\_default\_d. 'j\_behavior\_default\_o4 . 'j\_behavior\_default\_o12.J\_Behavior\_Default\_12$

$J\_Behavior\_Default\_5 \stackrel{def}{=}$

> $j\_behavior\_default\_a. 'j\_behavior\_default\_o4 . 'j\_behavior\_default\_o5.J\_Behavior\_Default\_4$

$+ j\_behavior\_default\_b .\ 'j\_behavior\_default\_o5 .\ 'j\_behavior\_default\_o7 . J\_Behavior\_Default\_7$

$+ j\_behavior\_default\_c .\ 'j\_behavior\_default\_o1 .\ 'j\_behavior\_default\_o5 . J\_Behavior\_Default\_1$

$+ j\_behavior\_default\_d .\ 'j\_behavior\_default\_o5 .\ 'j\_behavior\_default\_o13 . J\_Behavior\_Default\_13$

$J\_Behavior\_Default\_6 \stackrel{def}{=}$

$\qquad j\_behavior\_default\_a .\ 'j\_behavior\_default\_o6 .\ 'j\_behavior\_default\_o7 . J\_Behavior\_Default\_7$

$+ j\_behavior\_default\_b .\ 'j\_behavior\_default\_o4 .\ 'j\_behavior\_default\_o6 . J\_Behavior\_Default\_4$

$+ j\_behavior\_default\_c .\ 'j\_behavior\_default\_o2 .\ 'j\_behavior\_default\_o6 . J\_Behavior\_Default\_2$

$+ j\_behavior\_default\_d .\ 'j\_behavior\_default\_o6 .\ 'j\_behavior\_default\_o14 . J\_Behavior\_Default\_14$

$J\_Behavior\_Default\_7 \stackrel{def}{=}$

$\qquad j\_behavior\_default\_a .\ 'j\_behavior\_default\_o6 .\ 'j\_behavior\_default\_o7 . J\_Behavior\_Default\_6$

$+ j\_behavior\_default\_b .\ 'j\_behavior\_default\_o5 .\ 'j\_behavior\_default\_o7 . J\_Behavior\_Default\_5$

$+ j\_behavior\_default\_c .\ 'j\_behavior\_default\_o3 .\ 'j\_behavior\_default\_o7 . J\_Behavior\_Default\_3$

$+ j\_behavior\_default\_d .\ 'j\_behavior\_default\_o7 .\ 'j\_behavior\_default\_o15 . J\_Behavior\_Default\_15$

$J\_Behavior\_Default\_8 \stackrel{def}{=}$

$\qquad j\_behavior\_default\_a .\ 'j\_behavior\_default\_o8 .\ 'j\_behavior\_default\_o9 . J\_Behavior\_Default\_9$

$+ j\_behavior\_default\_b .\ 'j\_behavior\_default\_o8 .\ 'j\_behavior\_default\_o10 . J\_Behavior\_Default\_10$

$+ j\_behavior\_default\_c .\ 'j\_behavior\_default\_o8 .\ 'j\_behavior\_default\_o12 . J\_Behavior\_Default\_12$

$+ j\_behavior\_default\_d .\ 'j\_behavior\_default\_o0 .\ 'j\_behavior\_default\_o8 . J\_Behavior\_Default\_0$

$J\_Behavior\_Default\_9 \stackrel{def}{=}$

$\qquad j\_behavior\_default\_a .\ 'j\_behavior\_default\_o0 .\ 'j\_behavior\_default\_o9 . J\_Behavior\_Default\_8$

$+ j\_behavior\_default\_b .\ 'j\_behavior\_default\_o9 .\ 'j\_behavior\_default\_o11 . J\_Behavior\_Default\_11$

$+ j\_behavior\_default\_c .\ 'j\_behavior\_default\_o9 .\ 'j\_behavior\_default\_o13 . J\_Behavior\_Default\_13$

$+ j\_behavior\_default\_d .\ 'j\_behavior\_default\_o1 .\ 'j\_behavior\_default\_o9 . J\_Behavior\_Default\_1$

$J\_Behavior\_Default\_10 \stackrel{def}{=}$

$\qquad j\_behavior\_default\_a .\ 'j\_behavior\_default\_o10 .\ 'j\_behavior\_default\_o11 . J\_Behavior\_Default\_11$

$+ j\_behavior\_default\_b .\ 'j\_behavior\_default\_o8 .\ 'j\_behavior\_default\_o10 . J\_Behavior\_Default\_8$

$+ j\_behavior\_default\_c .\ 'j\_behavior\_default\_o10 .\ 'j\_behavior\_default\_o14 . J\_Behavior\_Default\_14$

$+ j\_behavior\_default\_d. 'j\_behavior\_default\_o2 . 'j\_behavior\_default\_o10.J\_Behavior\_Default\_2$

$J\_Behavior\_Default\_11 \overset{def}{=}$

$j\_behavior\_default\_a. 'j\_behavior\_default\_o10 . 'j\_behavior\_default\_o11.J\_Behavior\_Default\_10$

$+ j\_behavior\_default\_b. 'j\_behavior\_default\_o9 . 'j\_behavior\_default\_o11.J\_Behavior\_Default\_9$

$+ j\_behavior\_default\_c. 'j\_behavior\_default\_o11 . 'j\_behavior\_default\_o15.J\_Behavior\_Default\_15$

$+ j\_behavior\_default\_d. 'j\_behavior\_default\_o7 . 'j\_behavior\_default\_o11.J\_Behavior\_Default\_7$

$J\_Behavior\_Default\_12 \overset{def}{=}$

$j\_behavior\_default\_a. 'j\_behavior\_default\_o12 . 'j\_behavior\_default\_o13.J\_Behavior\_Default\_13$

$+ j\_behavior\_default\_b. 'j\_behavior\_default\_o12 . 'j\_behavior\_default\_o14.J\_Behavior\_Default\_14$

$+ j\_behavior\_default\_c. 'j\_behavior\_default\_o8 . 'j\_behavior\_default\_o12.J\_Behavior\_Default\_8$

$+ j\_behavior\_default\_d. 'j\_behavior\_default\_o4 . 'j\_behavior\_default\_o12.J\_Behavior\_Default\_4$

$J\_Behavior\_Default\_13 \overset{def}{=}$

$j\_behavior\_default\_a. 'j\_behavior\_default\_o12 . 'j\_behavior\_default\_o13.J\_Behavior\_Default\_12$

$+ j\_behavior\_default\_b. 'j\_behavior\_default\_o13 . 'j\_behavior\_default\_o15.J\_Behavior\_Default\_15$

$+ j\_behavior\_default\_c. 'j\_behavior\_default\_o9 . 'j\_behavior\_default\_o13.J\_Behavior\_Default\_19$

$+ j\_behavior\_default\_d. 'j\_behavior\_default\_o5 . 'j\_behavior\_default\_o13.J\_Behavior\_Default\_5$

$J\_Behavior\_Default\_14 \overset{def}{=}$

$j\_behavior\_default\_a. 'j\_behavior\_default\_o14 . 'j\_behavior\_default\_o15.J\_Behavior\_Default\_15$

$+ j\_behavior\_default\_b. 'j\_behavior\_default\_o12 . 'j\_behavior\_default\_o14.J\_Behavior\_Default\_12$

$+ j\_behavior\_default\_c. 'j\_behavior\_default\_o10 . 'j\_behavior\_default\_o14.J\_Behavior\_Default\_10$

$+ j\_behavior\_default\_d. 'j\_behavior\_default\_o6 . 'j\_behavior\_default\_o14.J\_Behavior\_Default\_6$

$J\_Behavior\_Default\_15 \overset{def}{=}$

$j\_behavior\_default\_a. 'j\_behavior\_default\_o14 . 'j\_behavior\_default\_o15.J\_Behavior\_Default\_14$

$+ j\_behavior\_default\_b. 'j\_behavior\_default\_o13 . 'j\_behavior\_default\_o15.J\_Behavior\_Default\_13$

$+ j\_behavior\_default\_c. 'j\_behavior\_default\_o11 . 'j\_behavior\_default\_o15.J\_Behavior\_Default\_11$

$+ j\_behavior\_default\_d. 'j\_behavior\_default\_o7 . 'j\_behavior\_default\_o15.J\_Behavior\_Default\_7$

References

Alur, R., R.K. Brayton, T.A. Henzinger, S. Qadeer and S.K. Rajamani. "Partial Order Reduction in Symbolic State Space Exploration." *Proceedings of the Ninth International Conference on Computer-aided Verification* (*CAV 1997*), Lecture Notes in Computer Science **1254**, Springer-Verlag, 1997. Pages 340-351.

Auletta, Richard J. "VHDL synthesized CSP Systems," *VHDL International Users Forum*, Fall 1991. Pages 95-102.

Arun-Kumar, S. and Matthew Hennessy. "An Efficiency Preoreder for Proceses." *Acta Informatica* **29**, 1992. Pages 737-760

Arun-Kumar, S. and V. Natajaran. "Conformance: A Precongruence close to Bisimilarity." *International Workshop on Structures in Concurrency Theory* (*STRICT '95*), J. Desel, ed., Workshops in Computing, Springer-Verlag, May 1995. Pages 55-68.

Bloom, B., S. Istrail and A. R. Meyer. "Bisimulation Can't Be Traced: Preliminary Report," *15th ACM Symposium on Principles of Programming Languages* (POPL), pp. 229-239, San Diego CA. 1988.

Brookes, S. D., C. A. R. Hoare and A. W. Roscoe. "A Theory of Communicating Sequential Processes," *JACM* **31**(3), pp. 560-599. 1984.

Burch, J. "Combining **CTL**, TraceTheory and Timing Models." *Automatic Verification Methods for Finite State Systems: Proceedings of the First CAV*, Lecture Notes in Computer Science **407**, pages 197--212. Springer-Verlag, 1989.

Cleaveland, Parrow and Steffin. *The Concurrency Workbench: A Semantics-based Verification Tool for Finite-state Machines*, Lecture Notes in Computer Science 407, Springer-Verlag, 1989.

Cleaveland, Rance and Joachim Parrow. "The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems," *ACM Transactions on Programming Languages and Systems*, **15**, no. 1, January 1993. Pages 36-72.

Corradini, F., R. Gorrieri and M. Roccetti. "Performance preorder and competitive equivalence," *Acta Informatica*, **34**, 1997. Pages 805-835.

Degano, Pierpaolo and Corrado Priami. "Non-interleaving semantics for mobile processes". *Theoretical Computer Science* **216,** 1999. Pages 237-270

De Nicola, R. and M. Hennessy. "Testing Equivalences for Processes," *Theoretical Computer Science* **34**, pp. 83-133. 1984.

Dijkstra, E. W., "Cooperating Sequential Processes," 43-112. *Programming Languages*, F. Genys (editor). Academic Press, New York. 1968.

Dukes, Michael , Frank M. Brown and Joanne E. DeGroat. "Verification of Layout Descriptions Using GES," 63-72. *Proceedings of the VHDL Users Group Spring 1991 Conference*. Menlo Park: Conference Management Services; 8-10 April 1991.

Dukes, Michael. *Hardware Verification Through Logic Extraction*. PhD Dissertation. AFIT. School of Engineering. Wright-Patterson Air Force Base OH; 1993.

Davis, Clarke and Stevens. "Automatic Synthesis of Fast Compact Asynchronous Control Circuits," Research Report No. 92/495/33. University of Calgary. Department of Electrical and Computer Engineering. Calgary, Alberta, Canada. 1992.

DoD (United States Department of Defense). *Standard General Requirements for Electronic Equipment*. MIL-STD-454N. Requirement 64 (Microelectronic Devices). Washington: Government Printing Office, 1992.

Fernandez, Jean-Claude. "An Implementation of an Efficient Algorithm for Bisimulation Equivalence", *Science of Computer Programming* **13** 219-236. Elsevier Science. 1989.

Fujita, Mashahiro, Hidehiko Tanaka and Tohru Moto-oka. "Temporal Logic Based Hardware Description and Its Verification with Prolog," *New Generation Computing* **1** (1983) 195-203. OHMSHA, Ltd. and Springer-Verlag. 1983.

Fujita, Mashahiro, Hidehiko Tanaka and Tohru Moto-oka. "Verification with Prolog and Temporal Logic," *Computer Hardware Description Languages and their Applications*, T. Uehara and M. Barbacci (Editors). North-Holland Publishing Company. © IFIP 1983.

Godefroid, Patrice**.** *Partial Order Methods for the Verification of Concurrent Systems.* Doctoral Thesis. University of Liege. 1995.

Gordon, Andrew D. "A Tutorial on Coinduction and Functional Programming." *Proceedings of the 1994 Glasgow Workshop on Functional Programming*. Springer Workshops on Computing. 1995.

Gordon, Michael. *The HOL Manual*. 1987.

-----. *The HOL System Tutorial*. Cambridge Research Center of SRI International under a grant from DSTO Australia, 8 December 1989.

Groote, J. F. and F. W. Vaandrager. *Structured Operational Semantics and Bisimulation as a Congruence*. Report CS-R8845, Centrum voor Wiskunde En Informatica, Amsterdam. 1988.

Guttag, John V. and James J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag. New York. 1993.

Hennessy, Matthew. *Algebraic Theory of Processes*. MIT Press. Cambridge MA. 1988.

Hennessy, M. and R. Milner. "Algebraic Laws for Nondeterminism and Concurrency," *JACM* **32**(1), pp. 137-161. 1985.

Hoare, C. A. R. "Communicating Sequential Processes," *On the Construction of Programs—an Advanced Course* (R. M. McKeag and A. M. Macnaghten, eds.), pp. 229-254. Cambridge University Press. 1980.

-----, *Communicating Sequential Processes*. Prentice Hall International, London, 1985.

Hua, Gary Xin and Hantoa Zhang. "Formal Semantics of VHDL for Verification of Circuit Designs. *1993 IEEE International Conference on Computer Design*. IEEE Computer Society Press, Los Alamitos CA. 1993.

IEEE (Institute of Electrical and Electronics Engineers). *IEEE Standard for Waveform and Vector Exchange (WAVES),* IEEE Std. 1029.1-1991, IEEE Press, New York (1991).

-----. *IEEE Standard VHDL Language Reference Manual*, IEEE Press, New York (1993).

-----. *IEEE P1364.1/Draft 2.0*, February 11,2002.

Ingólfsdóttir, Anna and Andrea Schalk. *A Fully Abstract Denotational Model for Observational Congruence*. Basic Research in Computer Science Report BRICS-RS-95-40, August 1995.

Jacobs, Bart and Jan Rutten. "A Tutorial on (Co)Algebras and (Co)Induction." *Bulletin of EATCS* **62**, pages 222-259. 1997

Jamsik, Damir and Mark Bickford. *Formal Verification of VHDL Models*. Final Technical Report RL-TR-94-3. Rome Laboratory. Air Force Material Command. Griffiss Air Force Base NY.

Lipsett, Roger, Schaefer, Carl and Ussery, Cary. *VHDL: Hardware Description and Design*. Springer-Verlag. 1989.

Liu, Ying. *Reasoning about Asynchronous Designs in CCS.* Master's Thesis. University of Calgary. Department of Electrical and Computer Engineering. Calgary, Alberta, Canada. 1992.

Lüttgen, Gerald and Walter Vogler. *A Faster-than Relation for Asynchronous Processes.* ICASE Report No. 2001-2. NASA Langley Research Center, Hampton VA. January 2001

Maclane, Saunders and Garrett Birkhoff. *Algebra*. Chelsea Publishing Company, New York, 1993.

Manna, Z. and A Pnueli. *The Temporal Logic of Reactive Systems: Specification,* Springer-Verlag. 1992.

Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers. Boston (1993).

Milner, R. "Calculi for Synchrony and Asynchrony," *Theoretical Computer Science* **25**, pp. 267-310. 1983.

Milner, R. *Communication and Concurrency*, Prentice Hall. New York, London. 1989.

Musgrave, Gerry, Roger B. Hughes and David Duncombe. "Review of Verification Techniques." http://www.ahl.co.uk. 1997.

Noh, Tim H. "Microcircuit Quality and Design Verification Improvement Project." Defense Electronics Supply Center presentation to the Air Force Producibility, Reliability, Availability and Maintainability (PRAM) office, Wright-Patterson Air Force Base OH.

Olderog, E. R. and C. A. R. Hoare. "Specification-oriented Semantics for Communicating Processes," *Acta Informatica* **23**, pp. 9-66. 1986.

Park, D. M. R. "Concurrency and Automata on Infinite Processes," *Proceedings 5th GI Conference* (P. Deussen, ed.) LNCS 104, pp. 167-183. Springer-Verlag, 1981.

Perry, William. "Memorandum for Secretaries of the Military Departments," 29 June 1994.

Phillips, I. C. C. "Refusal Testing," *Theoretical Computer Science* **50**, pp. 241-284. 1987.

Read, Simon and Martyn Edwards. "A Formal Semantics of VHDL in Boyer-Moore Logic." *Proceedings of the International Conference on Concurrent Engineering and Electronic Design Automation (CEEDA '94)* (Medhat, ed.). Department of Computation, UMIST Manchester, Great Britain.

Rounds, W. C. and S. D. Brookes. "Possible Futures, Acceptances, Refusals and Communicating Processes," *Proceedings 22nd Annual Symposium on Foundations of Computer Science*, pp. 140-149. IEEE. New York. 1981.

Rutten, J. J. M. M. *Universal Coalgebra: a Theory of Systems*. Report CS-R9652, Computer Science/Department of Software Technology, Centrum voor Wiskunde en Informatica. Amsterdam, The Netherlands. 1996.

Segala, Roberto. *A Process Algebraic View of I/O Automata*. MS Thesis. Massachusetts Institute of Technology, Cambridge MA. 1994.

Shams, M., J. C. Ebergen and M. I. Elmasry. "Modeling and Comparing CMOS Implementations of the C-Element," *IEEE Transactions on VLSI Systems* **6(4)**, pp. 563-567. December 1998.

Seitz, Charles. "System Timing." *Introduction to VLSI Design* (Mead and Conway). 1980.

Stevens, K. S. *Practical Verification and Synthesis of Low Latency Asynchronous Systems*. Doctoral Dissertation. The University of Calgary. Calgary, Alberta, Canada. 1994.

Stevens, K. S., J. Aldwinckle, G. Birtwistle and Y. Liu. "Designing Parallel Specifications in CCS." *1993 Canadian Conference on Electrical and Computer Engineering*, **II**, pages 983-6. Vancouver. Sepetember 1993.

Stirling, Colin, *Modal and Temporal Logics for Processes.* Technical Report ECS-LFCS-92-221, Laboratory for the Foundations of Computer Science, Computer Science Department, University of Edinburgh, 1992.

Wegner, Peter and Dina Goldin. *Mathematical Models of Interactive Computing*. Technical Report, Brown University, Jan 1999.

Wolper, Pierre. "Temporal Logic Can Be More Expressive." *CH1695-6/81/0340$00.75*. IEEE. 1981.

van Glabbeek, R. J. The Linear Time - Branching Time Spectrum. Technical Report CS-R9029, Centre for Mathematical and Computer Science, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands, 1990.

-----. "The Linear Time - Branching Time Spectrum," *CONCUR '90*

van Tassel, John P. *Femto-VHDL: The Semantics of a Subset of VHDL and Its Embedding in the HOL Proof Assistant*. Doctoral Dissertation. Cambridge University, July 1993.

Vita

Ronald W. Brower earned the Bachelor of Science in Physics and Mathematics from Wichita State University in 1967 and a second Bachelor of Science degree in Computer Science from Wright State University in 1985. He earned the Master of Science degree from the Air Force Institute of Technology in 1986.

While in the U.S. Army, he served a tour at the Electronic Devices and Technology Laboratory, Ft. Monmouth, New Jersey. After leaving the Army, he was a process development engineer for NCR's Microelectronics Division in Miamisburg, Ohio, where he was granted six patents. He then returned to government service and was an engineering supervisor for the Defense Electronics Supply Center in Dayton, Ohio. When that Center was closed in 1996, he transferred to the Aeronautical Systems Center at Wright-Patterson AFB, Ohio, where he worked as an avionics engineer for F-22 Program. He is presently a researcher for the Information Directorate at the Air Force Research Laboratory at Wright-Patterson.

Permanent Address:

Air Force Research Laboratory
Information Directorate, AFRL/IFTA
2241 Avionics Circle, Bldg. 620
Wright-Patterson AFB, OH 45433-7334
Tel: (937)255-6548x3590
Email: Ronald.Brower@wpafb.af.mil

REPORT DOCUMENTATION PAGE

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of the collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to an penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE (DD-MM-YYYY) 09-2002	2. REPORT TYPE **Doctoral Dissertation**	3. DATES COVERED (From – To) Dec 1997-Aug 2002

4. TITLE AND SUBTITLE

CONGRUENT WEAK CONFORMANCE

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Brower, Ronald W., DR-II, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 P Street, Building 640
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/DS/ENG/02-04

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

N/A

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This research addresses the problem of verifying implementations against specifications through an innovative logic approach. *Congruent weak conformance*, a formal relationship between agents and specifications, has been developed and proven to be a congruent partial order. This property arises from a set of relations called *weak conformations*. The largest, called *weak conformance*, is analogous to Milner's *observational equivalence*. Weak conformance is not an equivalence, however, but rather an ordering relation among processes. Weak conformance allows behaviors in the implementation that are unreachable in the specification. Furthermore, it exploits output concurrencies and allows interleaving of extraneous output actions in the implementation. Finally, reasonable restrictions in CCS syntax strengthen weak conformance to a congruence, called *congruent weak conformance*. At present, congruent weak conformance is the best known formal relation for verifying implementations against specifications. This precongruence derives maximal flexibility and embodies all weaknesses in input, output, and no-connect signals while retaining a fully replaceable conformance to the specification.

Congruent weak conformance has additional utility in verifying transformations between systems of incompatible semantics such as found in circuit development, security system design, and software engineering. This dissertation describes a hypothetical translator from the informal simulation semantics of VHDL to the bisimulation semantics of CCS. A second translator is described from VHDL to a broadcast-communication version of CCS. By showing that they preserve congruent weak conformance, both translators are verified.

15. SUBJECT TERMS

Asynchronous Systems, Automata, Bisimulation, CCS, Concurrency, Congruence, Digital Systems, Formal Methods, Precongruence, Preorder, Process Algebra, Semantics, Simulation, Specifications, Verification, VHDL.

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON Gary B. Lamont, Professor, AFIT/ENG
a. REPORT	b. ABSTRACT	c. THIS PAGE	UU	199	19b. TELEPHONE NUMBER (Include area code) (937) 255-3636, ext 4718; e-mail: Gary.Lamont@afit.edu
U	U	U			