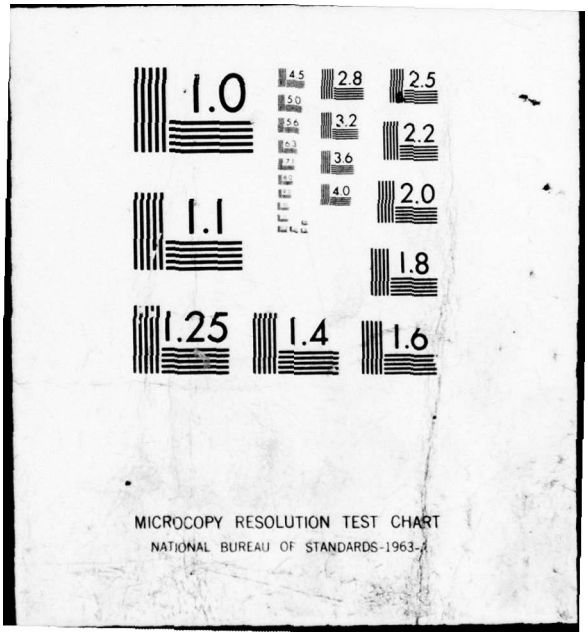
AD-A050 154 SRI INTERNATIONAL MENLO PARK CALIF F/G 9/2
THE SEMIAUTOMATIC GENERATION OF INDUCTIVE ASSERTIONS FOR PROVIN--ETC(U)
NOV 77 B ELSPAS, R S BOYER, K N LEVITT F44620-73-C-0068
AFOSR-TR-78-0114 NL

UNCLASSIFIED

1 OF 2
AD
A050 154


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

THE SEMIAUTOMATIC GENERATION OF INDUCTIVE ASSERTIONS FOR PROVING PROGRAM CORRECTNESS

Interim Report

Covering the Period 1 July 1974 through 30 June 1977

November 1977

By: Bernard Elspas, Principal Investigator
Robert S. Boyer
Karl N. Levitt
J Strother Moore
Lawrence Robinson
Robert E. Shostak
Jay M. Spitzen

Prepared for:

Air Force Office of Scientific Research
Bolling Air Force Base, D.C. 20332
Attn: Lt. Col. George W. McKemie, Contract Monitor
Directorate of Mathematical and Information Sciences

Contract F44620-73-C-0068

SRI Project 2686

SRI International
333 Ravenswood Avenue
Menlo Park, California 94025
Cable: STANRES, Menlo Park
TWX: 910-373-1246

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(19) **REPORT DOCUMENTATION PAGE**

READ INSTRUCTIONS BEFORE COMPLETING FORM

(18) 1. REPORT NUMBER: AFOSR-TR-78-0114

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

(6) 4. TITLE (and Subtitle): THE SEMIAUTOMATIC GENERATION OF INDUCTIVE ASSERTIONS FOR PROVING PROGRAM CORRECTNESS.

(9) 5. TYPE OF REPORT & PERIOD COVERED: Interim rept. 1 Jul 74-30 Jun 77

6. PERFORMING ORG. REPORT NUMBER: SRI Project 2686

(10) 7. AUTHOR(s): Bernard Elspas, Robert S. Boyer, Karl N. Levitt, J. Strother Moore, Lawrence Robinson, [redacted]

8. CONTRACT OR GRANT NUMBER(s): (15) F44620-73-C-0068

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: (16) 61102F 2304 (17) A2

11. CONTROLLING OFFICE NAME AND ADDRESS:
Air Force Office of Scientific Rsch/NM
Bolling AFB, DC 20332

(11) 12. REPORT DATE: Nov [redacted] 77

13. NUMBER OF PAGES: (12) 126 p.

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

15. SECURITY CLASS. (of this report): UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software verification
Synthesis of inductive assertions
Subgoal induction
Generator induction
Hierarchical design methodology
Presburger decision algorithm

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This interim report describes progress over the period 1 July 1974 through 30 June 1977 on a project aimed ~~ultimately~~ at solving a serious problem that has been encountered in attempts to make program correctness proving a practical technique for software verification. The principal problem addressed here is the difficulty of synthesizing so-called loop assertions in connection with the main method now under study for program proving--the inductive assertion method of Floyd.

→ next page

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

410 281

20. Abstract

In the first year of this project (1973-1974), the main emphasis was on exploring the potential of an approach to semiautomatic synthesis of inductive assertions by mechanizing the solution of finite difference equations for each program loop. This approach was found to be useful, but limited essentially to numerical algorithms. During the past three years (1975-1977), we have explored alternatives that gave promise either of alleviating the difficulty of assertion synthesis or of eliminating the need for it. Several rather diverse approaches, some of them constituting such alternatives to the ~~Floyd~~ present technique, are considered here: transformation of programs into primitive recursive form before verification, the method of generator induction for proof of properties of complex data structures, the use of a hierarchical design methodology to structure programs so as to minimize the need for loop assertions, and methods related to subgoal induction and computational induction. The two latter methods were analyzed in detail and compared with the ~~Floyd~~ present approach to arrive at a better understanding of their mutual relationships.

In addition, we report on the development of two algorithms of great utility in connection with program correctness proving, in general, and with the interactive debugging of loop assertions, in particular. These are highly effective decision algorithms for certain (decidable) formula domains that appear frequently in correctness proofs. The first algorithm is for deciding validity of formulas in an extension of unquantified Presburger arithmetic where uninterpreted function symbols and predicate variables are allowed. The second algorithm operates on a subdomain of this first domain--equality formulas over unquantified Presburger arithmetic, but is considerably more efficient over this subdomain. Both algorithms have been implemented (in LISP). Two different versions of the first algorithm have been incorporated into an experimental program verifier (for JOVIAL programs) under a separate RADC contract.

SRI International

THE SEMIAUTOMATIC GENERATION OF INDUCTIVE ASSERTIONS FOR PROVING PROGRAM CORRECTNESS

Interim Report

Covering the Period 1 July 1974 through 30 June 1977

November 1977

By: Bernard Elspas, Principal Investigator
Robert S. Boyer
Karl N. Levitt
J Strother Moore
Lawrence Robinson
Robert E. Shostak
Jay M. Spitzen

Prepared for:

Air Force Office of Scientific Research
Bolling Air Force Base, D.C. 20332
Attn: Lt. Col. George W. McKemie, Contract Monitor
Directorate of Mathematical and Information Sciences

Contract F44620-73-C-0063

SRI Project 2686

Approved

Jack Goldberg, Director
Computer Science Laboratory

Earle D. Jones, Executive Director
Information Science and Engineering Division

ABSTRACT

This interim report describes progress over the period 1 July 1974 through 30 June 1977 on a project aimed ultimately at solving a serious problem that has been encountered in attempts to make program correctness proving a practical technique for software verification. The principal problem addressed here is the difficulty of synthesizing so-called loop assertions in connection with the main method now under study for program proving – the inductive assertion method of Floyd.

In the first year of this project (1973-1974), the main emphasis was on exploring the potential of an approach to semiautomatic synthesis of inductive assertions by mechanizing the solution of finite difference equations for each program loop. This approach was found to be useful, but limited essentially to numerical algorithms. During the past three years (1975-1977), we have explored alternatives that gave promise either of alleviating the difficulty of assertion synthesis or of eliminating the need for it. Several rather diverse approaches, some of them constituting such alternatives to the Floyd technique, are considered here: transformation of programs into primitive recursive form before verification, the method of generator induction for proof of properties of complex data structures, the use of a hierarchical design methodology to structure programs so as to minimize the need for loop assertions, and methods related to subgoal induction and computational induction. The two latter methods were analyzed in detail and compared with the Floyd approach to arrive at a better understanding of their mutual relationships.

In addition, we report on the development of two algorithms of great utility in connection with program correctness proving, in general, and with the interactive debugging of loop assertions, in particular. These are highly effective decision algorithms for certain (decidable) formula domains that appear frequently in correctness proofs. The first algorithm is for deciding validity of formulas in an extension of unquantified Presburger arithmetic where uninterpreted function symbols and predicate variables are allowed. The second algorithm operates on a subdomain of this first domain – equality formulas over unquantified Presburger arithmetic, but is considerably more efficient over this subdomain. Both algorithms have been implemented (in LISP). Two different versions of the first algorithm have been incorporated into an experimental program verifier (for JOVIAL programs) under a separate RADC contract.

PROJECT PERSONNEL

The following people constituted the project team for this effort at various times during the period 1 July 1974 through 30 June 1977:

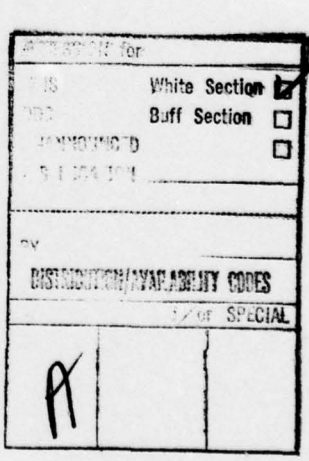
Robert S. Boyer
Bernard Elspas, Principal Investigator
Milton W. Green
Robert E. Shostak
Jay M. Spitzen

Other individuals whose names are listed as coauthors of this report, but whose efforts derived support entirely from contract sources apart from this contract, are:

Karl N. Levitt
J Strother Moore
Lawrence Robinson

Their names appear here as coauthors because of their collaboration on papers that have been included in the report.

The authors also wish to acknowledge helpful discussions with John Guttag, Ralph London, Zohar Manna, Mark Moriconi, Susan Owicki, Richard Waldinger, and Ben Wegbreit on various aspects of the work reported here.



CONTENTS

I AN OVERVIEW OF THE PROJECT RESEARCH

A. Introduction

This is an Interim Report covering progress during four years of research on problems allied to the generation of inductive assertions for program correctness proofs. Proof of correctness by mathematical techniques is one of the most promising approaches to the achievement of reliable computer programs--currently a source of major concern to the Air Force. In the principal method for carrying out such correctness proofs--the so-called Floyd method of inductive assertions--the program to be verified must first be supplied with formal mathematical assertions that hold at certain points throughout the program. Such inductive assertions are required in addition to input specifications and output specifications, which state the designer's intentions with respect to the overall behavior of the program. At least one inductive assertion is required for each control loop in the program.

In various current and recent research and development programs that are attempting to bring Floyd's method closer to practice, the invention of inductive assertions has been found to be a rather serious stumbling block to the application of that technique. The inductive assertions

must not only be valid for the program in question, but they must also be sufficiently powerful to permit the proof to succeed. In particular, each loop assertion must be powerful enough to imply itself around its loop, but usually no more powerful. It must also be deducible from foregoing assertions upon the first entry to that loop. Inventing loop assertions is rather like solving a complicated jigsaw puzzle where instead of physical pieces one has logical expressions that must be fitted together to make a consistent pattern according to the precise laws of rigorous formal logic. The problem is compounded by the fact that these "pieces" are not given a priori, but must be synthesized from more basic facts consistent with the laws of mathematics and program semantics. Considerable insight and ingenuity, as well as specialized knowledge of both verification in general and the specific program in hand, are required of the person engaged in verification. The project we report on here is an attempt to alleviate the difficulty of the assertion-synthesis aspects of program verification by the development of computer aids for the programmer/verifier. The hope is that, with the addition of interactive software tools to aid the programmer in synthesizing assertions and proving them, the verification process can be made a good deal more automatic.

B. Relation to Other Computer Science Laboratory Projects

The techniques developed thus far have been of appreciable benefit to progress on a related series of developmental projects that our group has been pursuing under contract with the Rome Air Development Center. These parallel contracts (F30602-75-C-0042 and F30602-76-C-0204) are entitled "Rugged Programming Environment" (RPE/1 and RPE/2). They have been concerned with the development of an experimental verification system for JOVIAL/J3 programs (see Elspas et al., 1976, 1977)*. The present AFOSR project has provided considerable theoretical support to this tool-building effort. A continuation of this development under RADC sponsorship is scheduled to begin in November 1977.

Mutually beneficial relationships have arisen also with several other government-supported projects in this laboratory. The first of these is an ONR-sponsored project (N00014-75-C-0816), which is concerned with the study of equivalence-preserving transformations between programs. For example, the initial work described below on formalisms for handling verification of programs entailing side effects has been continued under that ONR project. The second related project is an NSF grant (Number DCR74-18661) devoted to the development and study of a methodology for the hierarchical design and verification of programs. One of the papers (Spitzen, Levitt, and Robinson, 1976) describing

*References are listed at the end of the report.

work in that area was supported in part by these two projects as well as the AFOSR contract. A third project bearing some subject matter relationship to the present contract is NSF Grant Number DCR72-03737A01, which is concerned with mechanical theorem-proving techniques. The work of Boyer and Moore on their theorem prover for recursive functions is supported mainly by that grant.

C. An Overview of the First Year

We include here a brief summary of our first year's work (1973-74); details can be found in our first interim report (Elspas, 1974). During that first year, we developed a number of interactive software aids, the principal one being a semiautomatic generator for inductive invariants based on the method of difference equations. A good deal of insight was also gained into the fundamental principles entailed in the invention process. For example, an intimate relation was discovered between the invention of inductive assertions in the Floyd approach and the discovery of generalizations of a theorem to be proved in an alternate approach (Boyer-Moore) to program proving. Similarly, it was shown that for single-loop "while...do" programs one can always express a sufficient inductive assertion in a canonical form making use of logical quantifiers and the notion of the nth iterate of a function. While this was of

some theoretical interest, in practice there proved to be serious limitations to the use of this canonical form. It was shown that the Floyd method of proving partial correctness could readily be extended to include proofs of termination if one is permitted to modify the given program by adding what we then called loop-index variables (or loop counters). Essentially, these counters serve to record the number of times a loop has been traversed. To prove that the program will always terminate the verifier then needs to insert clauses into his assertions that bound these counters above by some function of the input data (see Elspas, 1974). Finally, the difference equation method was found to be moderately useful for numerical algorithms, but of very little utility for nonnumerical data processing.

D. Overview of the Second through Fourth Years

Beginning with the second year of research (1974-75), we spent considerably less time in tool building and proportionately more in exploring alternative approaches to the synthesis of inductive assertions. The aim in so doing was to attempt to uncover more basic relationships that might lead to a better understanding of the problem and eventually permit the implementation of more effective tools. The following avenues have constituted the main lines of our investigations during the period 1974-77:

* Transformation of programs to primitive recursive forms.

* Techniques for proving properties of complex data structures (principally the method of generator induction).

* Hierarchical design techniques.

* Investigations of the relations between such alternative approaches to verification as computational induction, subgoal induction, the schema approach of Basu and Misra, and the method of transformation into primitive recursive form.

In addition, substantial effort was devoted to the development of increasingly efficient decision algorithms for formulas in certain domains that we knew to be decidable (i.e., domains for which one can have a completely mechanical procedure for determining truth and falsity of formulas.) These domains are, in one form or another, extensions of the domain of unquantified Presburger formulas. There were two distinct motivations for this work. First, there was a real need for efficient decision algorithms in connection with our parallel development work for RADC on an experimental program verification system for JOVIAL. Second, we perceived that quick tests of validity (and of nonvalidity) would be extremely useful in the incremental synthesis of inductive assertions for test programs on which we could try out various heuristics. Several such algorithms were developed by Shostak. Two of

them have been implemented and were incorporated into the RPE/1 and RPE/2 JOVIAL verifiers. The algorithms are described in detail in Section III of this report.

We next describe briefly the nature of our recent work in the four areas listed above.

1. Transformation of Programs into Primitive Recursive Form

We began to investigate this approach during the second year, and the work was continued into 1976. Boyer had noted earlier that the invention of inductive assertions in Floyd's verification method was analogous to the so-called generalization step in the Boyer-Moore method for verifying recursive programs. When the recursive program appears in the special form of primitive recursion, this generalization step is often trivial. However, the direct translation of a flowchart program into recursive form does not usually lead to primitive recursion. This suggested the possibility of looking for techniques to permit the transformation of nonprimitive recursion schemas into primitive recursive form. During the second year Boyer and Shostak discovered several techniques for carrying out such a transformation for certain classes of recursion schemas. Their results are embodied in a paper with J Strother Moore (then with the

Xerox Palo Alto Research Center), which was presented at an ACM Symposium on Principles of Programming Languages in Atlanta, Ga., and which appears as an appendix to this report. It might be mentioned that Moore has pursued (independently of this project) a related notion, likewise growing out of the original Boyer-Moore LISP Theorem Prover. Moore's approach was to modify the LISP Theorem Prover to handle explicit program loops, but the underlying idea is to transform such loops mechanically into recursions (see, e.g., Moore, 1975). Both approaches (i.e., those of Boyer-Shostak and of Moore) were strongly motivated by the observation that verification tends to be much more straightforward for recursive functions than for general programs, since the latter generally require the inductive assertion method with its concomitant need for inventing loop assertions whereas, for programs that consist entirely of a system of mutual recursions, the recursive definitions themselves provide the equivalent of the otherwise needed inductive invariants. There are, of course, other difficulties inherent in the recursive approach, e.g., the problem of discovering the right generalizations, but at least they arise in a more controlled context. More uniform, systematic techniques of mathematical theorem proving are applicable to pure recursion, since the principal deduction tool is mathematical induction applied

to the expansion of recursive function definitions. The reader is directed to Appendix B for details of the process.

2. The Method of Generator Induction

A second alternative that we began pursuing during 1974-75 was an approach to the problem of proving properties of programs that create and manipulate complex data structures. The method that was devised (by Spitzen in collaboration with Wegbreit) was an extension of the notion of generator induction. One is concerned here with verifying that all procedures that are allowed to create or manipulate data structures of a given type (mode or class) will necessarily preserve the consistency requirements of that type. Proofs of this sort are becoming increasingly important in connection with such matters as the verification of security features of operating systems. Moreover, such proofs seem to be excessively awkward to carry out purely in terms of the Floyd inductive assertion approach. Generator induction, on the other hand, permits such proofs to be decomposed into small, comprehensible units corresponding to the program's structure. Such proofs emulate the desirable features of good informal proofs, wherein the proof is understandable (and believable!) because the program is partitioned into loosely coupled parts, for each of which some simple property is

demonstrated, and for which one shows that the parts are composed according to simple rules.

Details of the method are given in a paper (Wegbreit and Spitzen, 1976), which also relates generator induction to other proof techniques, principally structural induction (Burstall, 1969). A detailed example (an implementation of hashtables) is carried through in the paper, and several crucial properties of hashtables are proved by generator induction.

3. Hierarchical Design Techniques

Another alternative that we have investigated (although to a much lesser extent on this project than either primitive recursive transformation or generator induction) is that of hierarchical specification. The notion here is that much of the need for synthesizing inductive assertions should, in principle, be eliminated by proper program design from the start. Specifically, in the hierarchical design methodology (HDM) that has been developed (with other support) during the past few years by several staff members of our Computer Science Laboratory (see, e.g., Robinson and Levitt, 1977), program design is carried out through successive layers of abstraction, wherein abstract machines at each level are implemented in terms of modules at the

next lower level. At each abstraction level, the modules are first thoroughly specified in an assertion language as to their effects on the abstract data structures accessible to them. Relations between data structures at different levels are likewise characterized by mapping functions that can be precisely defined. The beneficial result of this systematic approach is that the proof process involved in verification of correctness by Floyd's approach can be partitioned into a series of proofs, each of which determines that an abstract module is correctly implemented in terms of lower-level modules, i.e., that the implementation of a module in terms of these lower modules is consistent with its specifications. This partitioning avoids much, but not all, of the need for inventing assertions, since many of the needed assertions already appear in the module specifications. Inductive assertions are still needed, however, whenever a module incorporates an explicit loop, a while...do statement, or the like.

On this project, we attempted to analyze the benefits of the hierarchical approach by working through a complex example. This was the hierarchical design and a formal implementation of a significant problem in the area of list processing: to efficiently maintain (i.e., create and access), lists so that no two are isomorphic ("hash-

consing"). This problem had been solved by L. P. Deutsch (1973) in his implementation of HCONS as part of a program verifier (PIVOT). The interest here was, of course, not in the solution per se, but in how the design methodology influences one's ability to prove its salient properties. The method of generator induction (discussed above) was used in part to verify these properties. The conclusions to be drawn from this example are that careful hierarchical structuring in program design leads to cleanly structured programs, and formal specification of program modules provides a precise guide to implementors (and their managers!). Most important, the partitioning that is inherent in the resulting programs leads to a corresponding partitioning of the proofs about these programs, as illustrated in the working example.

The results of this exercise are embodied in the paper by J. M. Spitzen, K. N. Levitt, and L. R. Robinson, entitled "An Example of Hierarchical Design and Proof," which appears in Appendix A. It has been submitted to the Communications of the ACM for publication in the Programming Languages Department of that journal. Dr. Spitzen's contribution to this paper was supported in part by the present contract.

4. Relations Among Alternative Approaches

During the past year of this research effort we began to investigate some relationships among several alternative ways of handling program correctness proving. Although this study is still in progress, it has provided some insights that we feel are useful. These preliminary results are contained in Section II of the report. In the next paragraphs we summarize the tentative conclusions to be drawn from that phase of our work.

During the period from roughly 1974 until the present, there has been no shortage of papers dealing with the problem of synthesizing inductive assertions and the closely related issues described above. From some of these papers one might conclude that the basic problem of assertion generation had been solved. A careful analysis of this literature, however, reveals that many of the most promising results actually depend heavily on additional assumptions that are not inherent in the problem. Moreover, several papers by different groups of authors present similar-sounding, but not identical, results that point at least to partial solutions. These papers fall into several groups, as follows:

* Papers dealing with elaborations of the difference equation technique, which we initiated in 1973.

* Those concerned with one or another sort of heuristic, or semiheuristic, techniques for deriving loop assertions from input and output specifications.

* More formal papers dealing in a strict mathematical manner with the basic question of determining for given classes of programs (whether in recursive or flowchart form) sets of either necessary, sufficient, or both necessary and sufficient conditions for correctness.

Briefly, the papers on difference-equation-related methods appear to pose no complete solution to the fundamental problem, a conclusion that we have previously asserted. However, systems for program verification can certainly gain a great deal by the incorporation of such features as difference equation solvers. It must merely be recognized that these features are not going to be of much help for nonnumerical programs.

Evaluation of the heuristic approaches poses a tougher problem. German and Wegbreit (1975) have pointed out that a great variety of essentially disjoint and complementary techniques are needed and that, even with all of these tools available and well implemented, much expertise is still required to carry through proofs for arbitrary programs. In addition to the difference-equation approach, the techniques covered there include:

* The method of "weak interpretation"

* The method predicate propagation

* The method of failure analysis (i.e., examining why a trial assertion fails to succeed, and trying to patch it up).

Even so, German and Wegbreit seem to feel that much more work is needed, and that "assertion synthesis at the level [they] believe desirable is still a distant goal." This is probably still true today (two years after they wrote their paper).

Among the more promising formal techniques are transformation into primitive recursive form (already described above), those proposed by Basu and Misra (1975), and those described by Morris and Wegbreit (1977). These approaches are all based on an analysis of schemas. Morris and Wegbreit have coined the term "subgoal induction" (and this name appears to have caught on). Basu and Misra (who preceded the other two authors in publication) did not choose to coin a special term for their approach. For want of a better name, we shall refer to it in the following simply as the "Basu-Misra approach."

The details of our comparison of these techniques are covered in Section II of this report.

E. An Overview of this Report

In Section II of this report, we present the essential results of a critical comparison of the approaches presented in the papers of Basu and Misra (1975) and Morris and Wegbreit (1977) with still earlier known techniques [principally, the method of computational induction, as developed by Manna and Pnueli (1969, 1970)]. We have also considered, within the same context, the method of transformation into primitive recursive form, but our conclusions regarding that approach are incomplete and are not presented here.

Section III of the report presents a detailed discussion of two algorithms developed by Shostak for deciding validity of formulas over certain decidable domains. An early version of the first algorithm described in Section III (for deciding formulas in an extension of unquantified Presburger arithmetic) was implemented in the RADC RPE/1 program verifier system in INTERLISP. A second, greatly improved version of this first algorithm was also first coded in INTERLISP and then translated into MACLISP prior to its incorporation into the RPE/2 verifier (for JOCIT). The second algorithm described in Section III is concerned with a more restricted formula domain, that of equality over Presburger arithmetic expressions augmented by

uninterpreted function symbols. For this subdomain it is more efficient than the first algorithm. It has not as yet been incorporated into any of our program verifiers. Presumably, it will be merged with the Presburger algorithm to increase the latter's effectiveness in dealing with functional equality formulas.

Section IV presents some general conclusions that we have arrived at in the course of these four years of research on the problem of program verification.

There are three appendices included in this report.

Appendix A contains the revised draft version of the paper by Spitzen, Levitt, and Robinson on "An Example of Hierarchical Design and Proof," which has been described briefly above in Section I-D-3.

Appendix B contains the results of our work on primitive recursive transformations, in the form of the paper, "Primitive Recursive Transformations," by R. S. Boyer, J S. Moore, and R. E. Shostak, which was delivered at the Third ACM Conference on Principles of Programming Languages, but which has not been published elsewhere.

Appendix C summarizes conference and workshop participation activities and publication of papers that were

supported wholly or partly through contract funds on this project.

II AN ANALYSIS OF RELATIONSHIPS AMONG SEVERAL ALTERNATIVE APPROACHES TO ASSERTION SYNTHESIS

A. Introduction

In this section we present, in summary form, the results of a careful comparison of several approaches to the synthesis of assertions. This analysis does not claim to be exhaustive, since we have omitted several important competitive techniques, e.g., the method of difference equations and various heuristic techniques (see German and Wegbreit, 1975). Our choice of approaches for this comparison was motivated by the observation that proving properties about recursively defined functions was usually more straightforward than the inductive assertion method (Floyd, 1967) used for flowchart programs. The obvious reason is that, in the recursive function case, the function definition itself can serve as the induction hypothesis (i.e., as the equivalent of the inductive assertion of the Floyd method). This same observation provided the motivation for our work on transformation of programs into primitive recursive forms (see Appendix B). It also provides much of the basis for the remarkable successes achieved by Boyer and Moore in mechanizing theorem proving for the domain of recursive functions (Boyer and Moore, 1975, 1977; Moore, 1975).

We have, therefore, selected for our analysis those techniques that derive, in one way or another, from the recursive point of view. Initially, the analysis focused on the method of subgoal induction (Morris and Wegbreit, 1977), with which we were acquainted from earlier oral presentations (circa January 1976), and some early, rather isolated unpublished results due to Basu and Misra on while-do schemas, which appeared to be related to subgoal induction. In the course of the analysis, however, it became apparent that the real genesis of both of these viewpoints lay in much earlier work by Manna and Pnueli (1969, 1970) on computational induction. However, the work in these three areas (i.e., computational induction, subgoal induction, and the Basu-Misra results) was couched in rather different terms and, moreover, often made different assumptions regarding such constraints as termination, "domain closure," and the "tightness" of the specifications to be proved. We felt it necessary, therefore, to rederive with some care the basic conclusions of the computational induction, subgoal induction, and Basu-Misra approaches.

In the following subsections, we attempt to present this unified viewpoint in a logically coherent manner with particular care given to the questions of termination, domain closure, and specification constraints. It is

difficult to summarize in one (or even a few) global theorems the massive detail involved, simply because the 'fine structure' of these proofs is often as important as the final results. However, since our ultimate aim is the automatic synthesis of loop assertions for the Floyd method, we should, perhaps cite as an overall result the theorem (Theorem A, below) due to Morris and Wegbreit, 1977. This theorem gives a sufficient condition on a while-do schema with input/output specifications for an adequate Floyd loop assertion to be mechanically constructed from the schema and its specifications. In their 1975 paper, Basu and Misra proved some related results (see Theorem B) specialized to the case of a functional output specification (i.e., where the result z of a while-do computation is specified to be z = G(x) in terms of the input variables). In addition, Basu and Misra have pointed out the centrality of notions of domain closure (roughly speaking, conditions under which intermediate results of the computation are guaranteed not to fall outside the domain specified by the input assertion), and also of some fine distinctions with respect to programs for which it is possible to construct an adequate loop invariant that does not refer to local variables of the loop.

B. Basic Assumptions and Notation

During the remaining discussion, we shall fix upon the recursive schema F shown below as prototypical for our purposes:

F: F(x) <= if ~B(x) then H(x) else L(x, F(N(x))).

We wish to determine conditions (both necessary and sufficient) under which this schema computes a function F(x) that satisfies an input specification PHI(x) and an output specification PSI(x, F(x)).

It will be assumed throughout that x ranges over a domain X, and that the functions H: X->X, N: X->X and L: XxX->X, as well as the predicate B: X->{true, false} are total. Observe, in particular, that for interpretations of F we may take x to be a vector <x[1],...,x[n]> of program variables.

Where we need to relate F to a flowchart program, we shall assume the trivial L given by L(x, z) = z, resulting in a (tail recursive) schema, which can be represented either in an equivalent while-do form or as a flowchart schema using a goto statement. Except where explicitly stated otherwise, F will refer to the general (recursive) schema shown above.

Free variables appearing in formulas are assumed to be universally quantified.

We use 'Term(F, x)' to mean "the computation of F terminates when it is initiated with input argument x."

The following facts derive entirely from the definition of F:

F1: ~B(x) -> Term(F, x) & F(x)=H(x)
F2: B(x) -> [Term(F, x) <-> Term(F, N(x))]
F3: B(x) & Term(F, x) -> F(x) = L(x, F(N(x)))

That is, F1-F3 are independent of any assumption that F satisfies the input/output specifications.

We shall make frequent use of the abbreviation:

R(x, z) = [PHI(x) -> PSI(x, z)]

C. Consistency with Specifications

We now define precisely what is meant by the consistency of the schema F with the specifications <PHI,PSI>. We need to distinguish two meanings depending on whether termination is assumed or not.

Definition: Consistency (strong sense)

F is said to be strongly consistent with respect to <PHI, PSI> when C is a valid formula:

C: PHI(x) -> Term(F, x) & PSI(x, F(x))

Definition: Partial Consistency (contingent on termination)

F is said to be partially consistent with respect to <PHI, PSI> when PC is valid:

PC: Term(F,x) -> R(x, F(x))

In the sequel we shall usually refer to PC as merely "consistency" (dropping the qualifier, "partial"). When termination is to be included in the requirement we shall use the term "strong consistency."

D. Consequences of the Definitions

We now list, with little or no proof, a series of formulas related to the consistency requirement PC, either by equivalence or implication (necessary or sufficient). In many cases, the derivations are achieved by relatively simple formula juggling. Where the derivations are much more complicated than this, we either indicate an original source for the result, or leave the proof to the reader. In this manner we avoid cluttering the argument with detail that would detract from the overall line of reasoning connecting PC with the final results. The logical connections among most of these formulas is shown diagrammatically in Section III-F.

C0 and C1, given below, are a pair of formulas whose joint validity is equivalent to PC:

```
C0:     ~B(x) -> R(x, H(x))
C1:     Term(F, x) & B(x) -> R(x, L(x, F(N(x)))).
```

A modified form of C1 is given by D1 below. D1 is weaker than C1, but [C0 & D1] is equivalent to [C0 & C1], and hence to PC.

```
D1:     Term(F,x) & B(x) & R(N(x), F(N(x))
            -> R(x, L(x, F(N(x)))).
```

D1 is the induction formula used in computational induction (Manna and Pnueli, 1969, 1970) for proofs of partial consistency. The proof that validity of C0 and D1 implies validity of C1 is achieved by an induction on the length of the computation, with C0 providing the base case.

E1, given below, is a stronger form of D1:

```
E1:     Term(F,x) & B(x) & R(N(x), z) -> R(x, L(x, z)).
```

E1 is essentially one of the subgoal induction VCs (stronger form) given in Morris and Wegbreit, (1977). We include termination as an explicit conjunct in the antecedent of E1, whereas this assumption is global in their paper. Note that z appears as a free variable in E1.

E1 can also be rewritten into the equivalent form:

```
Term(F,x) & PHI(x) & B(x) & [PHI(N(x)) -> PSI(N(x), z)]
        -> PSI(x, L(x,z)).
```

A weaker form of the subgoal VC, E1, is given by E1':

E1': Term(F,x) & PHI(x) & B(x) & PSI(N(x), z)
-> PSI(x, L(x,z)).

This is (essentially) the form of subgoal VC introduced and used most often by Morris and Wegbreit (1977).

E. Constraints on Specifications

The reader may have noted that so far some of the above formulas have been related only by unilateral implications. For example, E1 implies D1, but not conversely. Similarly, E1 implies E1', but not the converse.

The following restrictions on the specification R(x, z) [or, equivalently on PHI(x) and PSI(x, z)] are needed to establish the converse implications among the above forms of consistency conditions.

Definition: (Functionality) An output specification PSI(x,z) is said to have functional form if it can be written z = G(x), where G(x) is a mathematical function from X to X.

Definition: (Uniqueness) An output specification PSI(x,z) is said to possess uniqueness if

PSI(x, z1) & PSI(x, z2) -> z1 = z2

is valid.

Definition: (Well-behavedness) A specification <PHI, PSI> is said to be well-behaved (with respect to the schema F) when:

Forall x B(x) & PHI(x) -> Forsome z ~PSI(x, z)

is true.

Definition: (Tightness) A specification <PHI, PSI> is said to be tight (with respect to F) when both T1 and T2 hold:

T1. B(x) & PHI(x) & PSI(N(x), z1) & PSI(N(x), z2)
-> [L(x,z1)=L(x,z2)]

T2. F satisfies the Closure property (see below).

Definition: (Closure) The schema F is said to satisfy closure when

PHI(x) & B(x) -> PHI(N(x))

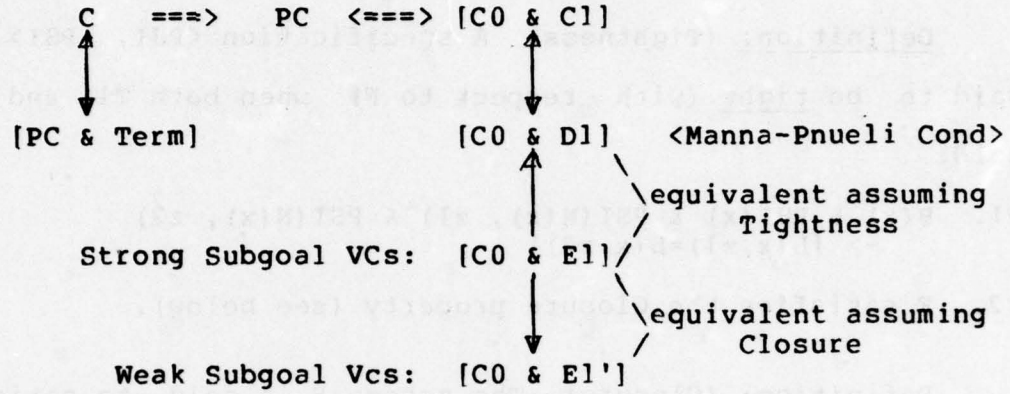
is valid. In this case, PHI(x) is a loop invariant of F.

Note: This notion appeared first in an early (unpublished) version of the Basu-Misra results. In the final version (Basu and Misra, 1975) it was also extended to a weaker notion. Closure also plays a central role in in the theory of subgoal induction.

F. Relations Among Conditions

By means of the diagrams below we show the implications that hold among the various versions of VCs listed above,

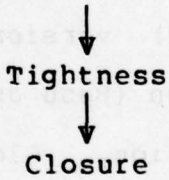
and the conditions under which these relations hold. The details of the proofs establishing these connections are not given here, but are planned for publication in a separate journal paper.



Some Subsidiary Relations:

Functionality ===> Uniqueness

Functionality & Closure ===> Uniqueness & Closure



Well-Behavedness & E1 ===> E1' & Closure ===> E1

Note: For the simple ('tail recursive') schema, where L(x, z) = z, one also has that uniqueness implies well-

behavedness, and therefore that functionality implies well-behavedness. These results are not valid for more general schemas such as F.

G. Relations To Floyd Verification Conditions

Our principal reason for analyzing the above relations is to establish the weakest conditions under which verification by means of inductive assertions (Floyd verification) will be feasible. In order that the Floyd method be applicable to schema F, we must restrict the output function L(x,z) to be simply z. For this case, we can replace the recursive schema F by an equivalent flowchart or while-do schema. An adequate inductive (loop) assertion is given by:

I(x,y) = PHI(x) & PHI(y) & Forall z [(PSI(y,z) -> PSI(x,z)],

provided the specifications are "tight" (as shown by Morris and Wegbreit, 1977).

The while-do schema WD incorporating all three assertions can then be written:

Schema WD:

```
begin
    assert PHI(x);
    y:= x;
    maintain I(x,y) while B(y) do y:= N(y);
    z:= H(y);
    assert PSI(x,z);
end,
```

where the inductive assertion has been inserted into the while statement by making use of the suggestive "maintain I" syntax introduced by Morris and Wegbreit (1977).

The Floyd VCs corresponding to this loop assertion I(x,y) are:

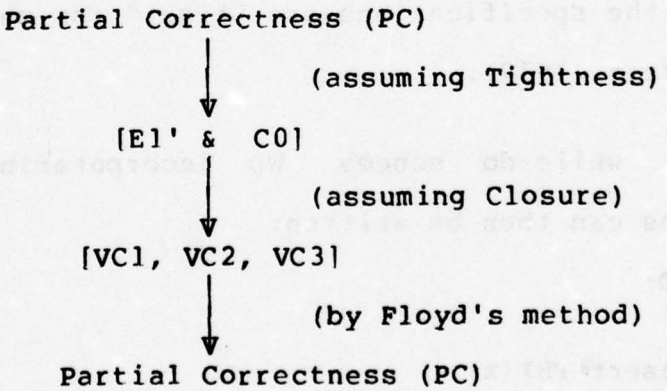
VC1: PHI(x) -> I(x,x) <trivially valid>

VC2: I(x,y) & ~B(y) -> PSI(x, H(y)) <follows from C0>

VC3: I(x,y) & B(y) -> I(x, N(y)) <follows from E1'>.

Conversely, if one can show directly that these VCs are valid, either for the given loop assertion I(x,y) or any other one, then Floyd's method proves partial correctness *(PC) for the schema.*

Diagramatically, we have the following implications:



A central conclusion that emerges from all of the foregoing analysis is captured by the following:

Theorem A: For the while-do schema WD, if the specifications <PHI,PSI> are tight, and WD is consistent (PC) with them, then

I(x, y) = PHI(x) & PHI(y) & Forall z [PSI(y,z) -> PSI(x,z)]

constitutes an adequate inductive assertion for demonstrating that consistency.

It is worth noting that the above inductive assertion takes a useful special quantifier-free form when the specification PSI is functional. For let PSI(x, z) be given by z = G(x). Then I(x, y) becomes:

PHI(x) & PHI(y) & Forall z [z=G(y) -> z=G(x)],

which is easily seen to be equivalent to:

PHI(x) & PHI(y) & [G(y) = G(x)].

Note that functionality implies that the uniqueness condition is satisfied, and this in turn implies the second tightness condition T2. This means that a functionally specified single-loop program that can be shown to satisfy domain closure can always be verified (if it is consistent with its specification R) by means of Floyd's method using

an extremely simple inductive assertion. These facts (which have been noted repeatedly by others) are summarized in the next theorem.

Theorem B: If the while-do schema WD is closed, and the specification PSI(x, z) has the functional form [z = G(x)], then the inductive assertion

$$I(x,y) = PHI(x) \;\&\; PHI(y) \;\&\; [G(y) = G(x)]$$

makes VC1-VC3 valid whenever the program satisfies its specifications (PC valid). This assertion is therefore adequate to prove partial correctness under the given assumptions.

There is another productive way to employ Theorem B, even when the given specification is not in functional form. Suppose that one can discover what function is "computed by the loop," regardless of the particular form PSI in which the output specification happens to be given. Suppose, moreover that this function can be expressed in some closed-form expression G(x). Then, by using Theorem B with the new output specification PHI*(x,z) = [z = G(x)], we observe that I(x,y) as given in Theorem B must be an adequate inductive assertion for proving PSI*(x,z) by Floyd's method. This reduces the proof of consistency relative to the original output assertion PHI to showing that the properties of G(x) imply PSI(x, G(x)).

Observe also that a separate proof is not needed to show that the loop output function has been accurately captured in the expression G(x). That fact will have been proved when the usual verification of the Floyd VCs is carried out with the invariant constructed in terms of G(x). This is merely one example of the general feature of Floyd-type verification whereby it is immaterial how the inductive assertions are obtained--one may even guess at them. The proof that they were right lies in the demonstration of validity of the VCs.

In the light of the preceding paragraph, one sees also that the difference-equation approach is simply one way of "wrapping up" an iterative (or recursive) loop into a closed-form expression. If it succeeds, the resulting closed-form solution may be used as G(x) in Theorem B to yield an inductive invariant.

This concludes our summary of the relations among VCs for computational induction (the Manna-Pnueli approach), subgoal induction (the Morris-Wegbreit approach), and the method of inductive assertions (Floyd approach).

H. Concluding Observations

It could be argued that our convention of using single variable names (e.g., x, y,...) to denote what might in

specific interpretations be vectors of program variables tends to obscure the fact that, in actual programs, the loop body may contain local variables. Indeed, Basu and Misra (1975) are emphatic about excluding such local variables from participation in the inductive assertion. Their viewpoint is a reasonable one for the do-while statement, but it loses its force when the program is written with gotos. Moreover, exclusion of local variables from the inductive assertion prevents certain programs from being verified by Floyd's method. Our convention, on the other hand, has been to permit local variables to appear in I(x,y) by regarding them as part of y. However, if y and x are to be vectors of the same dimensionality [as is assumed when we define N: D->D, where y=N(x)], this forces x to include all such local variables as well as true input parameters to F. These conventions mean that to prove correctness for a given three-parameter function Fl(u, v, w) that uses two local loop variables a and b which are initialized to, say, 0 and true in the loop body, we must first introduce a five-parameter auxiliary function F(u, v, w, a, b), where two of the parameters correspond to the local loop variables. Verification is then carried out for F(u, v, w, 0, true) = Fl(u, v, w). This forces one first to generalize the specifications on Fl to specifications about F. (Note, for example, that in Basu and Misra's version of the well-known

King-Floyd exponentiation program this generalization step has already been accomplished by the way they write the functional specification.) We do not feel that this need for generalization is more than a minor nuisance in practice. An equivalent process is required in, for example, the "generalization step" of the Boyer-Moore approach (Boyer and Moore, 1975). Moreover, our viewpoint permitting the use of such "locals" in the loop assertion allows us to verify programs such as Basu and Misra's example 9 (see Basu and Misra, 1975) by the Floyd method, whereas their restriction will preclude this, as they themselves state.

III DEDUCTION MECHANISMS FOR PROGRAM VERIFICATION

A. Introduction

In the past two years of work on this project, much emphasis has been placed on the development of efficient mechanical deductive techniques for specific domains. This emphasis has been motivated in part by the need for fast, automatic decision mechanisms in the RPE/2 work ("Rugged Programming Environment – RPE/2") under RADC sponsorship. Most of the deductive systems designed specifically for application to program verification have been of the heuristic, goal-driven type. The first SRI Program Verifier (written in QA4/QLISP) and much of the RPE/1 system were subgoaling systems depending heavily on ad hoc heuristics. While such systems are usually quite general and easy to modify, they tend to be unreliable, incomplete, and generally too slow to handle many of the larger, more complex verification conditions encountered in the RPE application in a reasonable period of time. Our early experience, dating back to the RPE/1 project (1975-76) and even before, indicated that a substantial fraction of the formulas actually encountered fall within domains that can be decided without need for heuristic methods. Accordingly, we initiated a course of research with the aim of producing fast, nonheuristic algorithms for these domains that could be easily implemented in the RPE/2 system.

The results of this effort are discussed in detail in the two subsections that follow. Section III-B describes a fast decision procedure for quantifier-free Presburger arithmetic augmented by uninterpreted function and predicate symbols. An INTERLISP version of this procedure was implemented in the latter part of 1975, and was tested extensively during the subsequent six months. A refined version of the procedure that more efficaciously handles formulas containing function symbols was developed and implemented (first in INTERLISP, later in MACLISP) during the fall of 1976. The refined version, detailed in Section III-B-7, was found to be a substantial improvement over its earlier counterpart, running two to three times faster on the same formulas.

Section III-C describes more recent work on an algorithm specifically intended for the class of unquantified equality formulas with function symbols. This class forms a proper subclass of unquantified Presburger formulas with function symbols and can, therefore, be decided using the earlier procedure. Owing to the restricted structure of

the subclass (inequalities are excluded), it was possible, however, to devise a decision algorithm for this subclass that usually operates much more quickly than the more general procedure. The new equality algorithm has not yet been implemented within the RADC Program Verifier for JOCIT, principally because translation from INTERLISP to MACLISP is required. We expect this transfer to take place within the coming phase (RPE/3) of our work for RADC, scheduled to begin in November 1977.

B. An Efficient Decision Procedure for Arithmetic with Function Symbols

1. Introduction

The procedure described here operates over an extension of the class of unquantified Presburger formulas. Briefly, Presburger formulas are those that can be built up from integers, integer variables, addition,* the usual arithmetical relations ($<$, $\leqslant$, $>$, $\geqslant$, $=$), and the first-order logical connectives. The formula $(\forall x)(Ey)\ 3x + y = 2 \supset x < y$, for example, falls within the class. The subclass of unquantified Presburger formulas consists of those Presburger formulas having no quantifiers.

The extension of unquantified Presburger we shall be dealing with introduces, for each $n \geqslant 0$, an unlimited number of n-ary function symbols (interpreted as functions from Z^n to Z) and n-ary predicate symbols (interpreted as relations over Z^n).

The formula

$$x < f(y) + 1 \wedge f(y) \leqslant x \supset (P(x,y) \equiv P(f(y),y)) \quad ,$$

for example, is a member of the extended class. One can easily check that this particular formula is valid (that is, it evaluates to true for all integers x, y, z, no matter what monadic integer function is assigned to f and dyadic integer relation to P).

Function symbols may appear in any term context and may have arbitrary terms as arguments, including expressions containing function symbols. For example, the formula $g(x + 2\ f(y)) = 4$ is a member of the class.

---

*Arbitrary multiplication is not permitted. It is convenient, however, to use multiplication by constants as an abbreviation for repeated addition; x + x + x is thus written 3x.

The extended theory includes a surprisingly large proportion of the formulas encountered in program verification. It is particularly well suited to programs that manipulate arrays and other data structures that can be modeled as uninterpreted functions. The semantics of the McCarthy (1962) ACCESS and CHANGE array primitives are easily encoded within the theory; for example, the formula

$$M = \text{ACCESS}(\text{CHANGE}(A,I,V), J + 2)$$

can be mechanically translated to an equivalent formula

$$J + 2 = I \supset M = V$$

$$\wedge\ J + 2 \neq I \supset M = A[J + 2] \quad ,$$

where A is now an uninterpreted function symbol.

A number of other constructs, including MAX, MIN, and ABSVALUE can be dealt with in a similar manner. For instance, the valid formula

$$x = y + 2 \supset \text{MAX}(x,y) = x$$

containing the interpreted function symbol MAX translates to

$$[x \geqslant y \supset \text{MAX}(x,y) = x \wedge y \leqslant x \supset \text{MAX}(x,y) = y]$$

$$\supset$$

$$[x = y + 2 \supset \text{MAX}(x,y) = x] \quad ,$$

where MAX is now interpreted.

The discussion that follows is presented in six subsections. Section III-B-2 provides historical perspective and cites related work; Section III-B-3 describes a decision method for the unextended class that forms the basis of the extended procedure. The next three subsections establish the decidability of the extended class and introduce a basic version of the procedure. The last subsection (III-B-7) presents an extremely efficient refinement of the basic procedure.

2. Related Work

The class of closed Presburger formulas was first shown to be decidable by M. Presburger (1929). The best-known decision procedure for it (described by Kreisel and Krevine, 1967, among others) is based on a method of elimination of variables. In its raw form, the algorithm is prone to combinatorial explosion, and is therefore not practical for nontrivial problems. More efficient versions of the method of elimination have since been given by D. C. Cooper (1971). Cooper's most recent procedure (Cooper, 1972) is the most efficient known algorithm for full Presburger. D. Oppen (1975) has shown that Cooper's algorithm is probably the best one can do in the worst case (deterministic time complexity on the order of $2^{2^{2^n}}$ in the length of the formula).

More recent work has focused on the subclass of Presburger without quantifiers. The decision complexity of this subclass is no worse than exponential, making it substantially easier to decide (in theory, at least) than full Presburger. A number of theorem provers for this class [Bledsoe (1975); Shostak (1977)] have been successfully implemented and used for program verification.

The extension of the unquantified subclass dealt with in this paper is also no worse than exponential deterministic time complexity. It is, perhaps, surprising that the incorporation of predicate and function symbols does not give away decidability altogether. Downey (1972) has proved that the addition of even a single monadic predicate symbol to the language of full Presburger produces a reduction class.

An implementation of our procedure (coded in INTERLISP for the DEC PDP-10) has been used for the past two years in conjunction with a system for verifying JOVIAL programs (Elspas et al, 1976). We have found that most formulas of a few lines are handled in seconds, and that larger formulas are generally decided much more quickly than by humans.

A number of other theorem provers dealing with similar theories are currently under development. The systems of N. Suzuki (1975) and of G. Nelson and D. Oppen are among these.

3. The Unextended Class

The new procedure can best be explained in relation to the author's adaptation (Shostak, 1977) of Bledsoe's (1975) method for handling the unextended unquantified class.

This method is carried out in two stages. In the first stage, the formula F to be decided is reduced to a set of integer linear programming problems (ILPs) with the property that F is valid if and only if none of the problems has a solution. In the second stage, the ILPs are tested one by one for solvability. If one is found to have an integer solution, the solution provides a model for $\neg F$ and therefore a conterexample for F.

Let us now consider these steps in greater detail.

The reduction to a set of ILPs consists of expanding the negation of F into a disjunctive normal form:

$$\neg F \equiv G_1 \vee G_2 \vee \ldots \vee G_p \quad ,$$

where each G_i is a conjunction of linear inequalities of the form $A \leqslant B$. During the expansion, terms of the form $A = B$ are replaced by $(A \leqslant B \wedge B \leqslant A)$. Similarly, $A \geqslant B$ is replaced by $B \leqslant A$; $A < B$ is replaced by $A + 1 \leqslant B$; $\neg(A \leqslant B)$ is replaced by $B + 1 \leqslant A$; and so on. The conjunctions $G_1, G_2, \ldots, G_p$ in the expanded form make up the set of ILPs. Suppose, for example, that

$$F \equiv (x < 3y + 2) \wedge x = 1 \supset x = y \quad .$$

Then

$$\begin{aligned}
\neg F &\equiv \neg[x < 3y + 2 \wedge x = 1 \supset x = y] \\
&\equiv x < 3y + 2 \wedge x = 1 \wedge \neg(x = y) \\
&\equiv x \leqslant 3y + 1 \wedge x \leqslant 1 \wedge x \geqslant 1 \wedge (x + 1 \leqslant y \vee y + 1 \leqslant x) \\
&\equiv (x \leqslant 3y + 1 \wedge x \leqslant 1 \wedge 1 \leqslant x \wedge x + 1 \leqslant y) \\
&\quad \vee (x \leqslant 3y + 1 \wedge x \leqslant 1 \wedge 1 \leqslant x \wedge y + 1 \leqslant x)
\end{aligned}$$

so $G_1 \equiv x \leqslant 3y + 1 \wedge x \leqslant 1 \wedge 1 \leqslant x \wedge x + 1 \leqslant y$

and $G_2 \equiv x \leqslant 3y + 1 \wedge x \leqslant 1 \wedge 1 \leqslant x \wedge y + 1 \leqslant x$.

$\neg F$ is satisfiable if and only if either G_1 or G_2 has a solution in integers, and so F is valid if and only if neither G_1 nor G_2 has such solutions. The G_i's are now tested for

feasibility by using either conventional integer programming algorithms [such as R. Gomory's (1958), or the SUP-INF method (Bledsoe, 1975; Shostak, 1977)].

Continuing the above example, it is easy to see that the ILP G_2 has the integer solution $x = 1$, $y = 0$. These values provide a model for $\neg F$ and hence a counterexample to F. (Counterexamples are also provided by G_1 which is integer feasible as well.)

4. Decidability of the Extended Class

The decision mechanism for the extended class elaborates upon a method for reducing an arbitrary formula F in this class to an equivalid formula $\hat{F}$ in the unextended class. The reduction is carried out in two steps, the first eliminating uninterpreted predicate symbols, and the second eliminating uninterpreted function symbols:

(1) For each uninterpreted n-ary predicate symbol P occurring in F, let f_p be a new n-ary function symbol. Obtain F′ from F by replacing each atomic formula $P(t_1, t_2, \ldots, t_n)$ by the formula $f_p(t_1, t_2, \ldots, t_n) = 0$.

(2) For each pair $f(t_1, t_2, \ldots, t_n)$, $f(u_1, u_2, \ldots, u_n)$ of distinct terms or subterms of terms in F′ with the same outermost uninterpreted function symbol f, construct the axiom:

$$t_1 = u_1 \wedge t_2 = u_2, \wedge \ldots \wedge t_n = u_n \supset f(t_1, t_2, \ldots, t_n) = f(u_1, u_2, \ldots, u_n)$$

Let F″ be the formula given by

$$A_1 \wedge A_2 \wedge \ldots \wedge A_r \supset F' \quad ,$$

where the A_is are the axioms so constructed. Next, for each term t occurring in F″ that has an uninterpreted outermost function symbol, let x_t be a new integer variable. Obtain $\hat{F}$ from F″ by replacing each such term t with x_t. (In the case where one such term is nested within another, the larger term is replaced.)

Consider, as an example, the valid formula F given below:

$$[(P(z) \supset z = 1) \wedge g(y) = z + 4] \supset [f(g(y)) = f(3 + 2z) \vee \neg P(1)] \quad .$$

Using Step (1) to eliminate the uninterpreted function symbol P, we obtain the formula F′ given by:

$$[f_p(z) = 0 \supset z = 1) \wedge g(y) = z + 4] \supset [f(g(y)) = f(3 + 2z) \vee f_p(1) \neq 0]$$

Applying Step (2), we observe that F′ contains two pairs of distinct terms with the same outermost function symbol – the pair $f_p(z)$, $f_p(1)$, and the pair $f(g(y))$, $f(3 + 2z)$. The formula F″ is therefore given by:

$$[z = 1 \supset f_p(z) = f_p(1)] \wedge [g(y) = 3 + 2z \supset f(g(y)) = f(3 + 2z)]$$

$$\supset$$

$$[(f_p(z) = 0 \supset z = 1) \wedge g(y) = z + 4] \supset [f(g(y)) = f(3 + 2z) \vee f_p(1) \neq 0] \quad .$$

Letting $f_p(z)$, $f_p(1)$, $g(y)$, $f(g(y))$, $f(3 + 2z)$ be replaced by x_1, x_2, x_3, x_4, and x_5, respectively, we obtain $\hat{F}$:

$$[z = 1 \supset x_1 = x_2] \wedge [x_3 = 3 + 2z \supset x_4 = x_5]$$

$$\supset$$

$$[(x_1 = 0 \supset z = 1) \wedge x_3 = z + 4] \supset [x_4 = x_5 \vee x_2 \neq 0] \quad .$$

This latter formula is contained within the unextended class, and can therefore be decided by using the method described in the last section.

The reduction just described is quite similar to W. Ackermann's (1954) method for eliminating function symbols from universally quantified equality formulas in predicate calculus with function symbols and identity. The correctness of the reduction can be proved straightforwardly; given a model for $\neg F$, one can construct a model for $\neg\hat{F}$, and conversely. The details are easily gleaned from Ackermann's proof, and so are omitted here.

While the reduction confirms the decidability of the extended class, it does not of itself provide a very good computational method. Recall that in Step (2) of the reduction, an axiom is constructed for each pair of terms, (including nested terms) with the same outermost uninterpreted function symbol. The number of such axioms is thus

proportional (in the worst case) to the square of the length of the given formula. Moreover, each axiom at least triples the number of ILPs that must be solved in deciding the reduced formula $\hat{F}$.

Suppose, for example, that the axiom $x = y \supset f(x) = f(y)$ is generated in Step (2). It is easy to check that the expansion into disjunctive normal form described in the last section produces three disjuncts (corresponding to the cases in which $x < y$, $x > y$, and $f(x) = f(y)$) for each disjunct that would have been produced in the absence of the axiom.

As an illustration of the kind of combinatorial explosion that can result, consider the formula F given by

$$x \leqslant g(x) \wedge x \geqslant g(x) \supset x = g(g(g(g(x)))) \quad .$$

An axiom must be generated for every pair of terms among g(x), g(g(x)), g(g(g(x))), and g(g(g(g(x)))). There are six such axioms, each one tripling the number of disjuncts appearing in the d.n.f. expansion of the corresponding reduced formula $\hat{F}$. Deciding $\hat{F}$ therefore entails the solution of 3^6 (= 729) ILPs.

In the event that the function symbol associated with a given axiom has more than one argument place, the combinatorial effect is even more pronounced; one can easily check that two more cases are developed by each additional argument position.

5. Basic Procedure for the Extended Class

The procedure given in this section greatly reduces the combinatorial explosion produced by the reduction process. The improvement is founded on two observations:

(1) In most cases, only a small part of the information contained in the generated axioms is of relevance to the validity of the reduced formula,

(2) It is frequently possible to determine which information is relevant in advance of its application.

The example formula $F \equiv x \leqslant g(x) \wedge g(x) \leqslant x \supset x = g(g(g(g(x))))$ of the last section serves well as an illustration of the basic idea. Suppose we pretend, for a moment, that F is a member of the unextended class, that is, that the terms g(x) and g(g(g(g(x)))) are simply integer variables that happen to have fancy names. If we then apply the procedure of Section III-B-3, the expansion into disjunctive form produces the two ILPs:

$$[x \leqslant g(x),\ g(x) \leqslant x,\ x \leqslant g(g(g(g(x)))) - 1] \quad \text{and}$$

$$[x \leqslant g(x),\ g(x) \leqslant x,\ g(g(g(g(x)))) \leqslant x - 1] \quad .$$

Let us focus on the first of these. If this ILP is solved, the following solution (among others) is obtained:

$$x = 0$$

$$g(x) = 0$$

$$g(g(g(g(x)))) = 1 \quad .$$

At this point, the procedure of Section III-B-3 terminates, offering the discovered solution as a counterexample to the formula F.

If we return to the view of F as a formula in the extended theory, however, it can be seen that the above solution is not a legitimate model for $\neg F$. In particular, the substitutivity axioms of equality forbid that x and g(x) be given the same value while g(g(g(g(x)))) is given a different value. (Note, incidentally, that this violation occurs in all solutions of the ILPs in question.) The violated substitutivity property is neatly expressed by the following formula:

$$x = g(x) \supset g(x) = g(g(g(g(x)))) \quad .$$

If we now assert this formula as a hypothesis of F, the following formula F* is obtained:

$$[x = g(x) \supset g(x) = g(g(g(g(x))))]$$

$$\supset$$

$$x \leqslant g(x) \wedge x \geqslant g(x) \supset x = g(g(g(g(x)))) \quad .$$

If the new formula F* is viewed as a member of the unextended class and given to the procedure of Section III-B-3, the resulting ILPs are found not to have any integer solutions. The original formula F must, therefore, be valid as a member of the unextended class and hence as a member of the extended class.

Note that, in the case of our example, this approach requires the solution of only seven ILPs (one to provide the illegitimate counterexample and six to decide the augmented formula F*), as opposed to the 729 required by the reduction method. Part of the improvement is attributable to the omission of unneeded axioms generated in the reduction method. More importantly, the three axioms from that method that <u>are</u> relevant, $(x = g(x) \supset g(x) = g(g(x))$, $g(x) = g(g(x)) \supset g(g(x)) = g(g(g(x)))$, and $g(g(x)) = g(g(g(x))) \supset g(g(g(x))) = g(g(g(g(x))))$ are replaced by a single formula:

$$x = g(x) \supset x = g(g(g(g(x)))) \quad .$$

This replacement alone accounts for a ninefold reduction in the number of ILPs that must be solved in the example problem.

We now give a detailed description of the procedure:

(1) Using Step (1) of the reduction method, all uninterpreted predicate symbols are eliminated from the formula F to be decided.

(2) Expressions involving + or * that occur as arguments to uninterpreted function symbols are eliminated through the introduction of new variables. [For example, the formula $x \leqslant y + f(3z + 5)$ becomes $z' = 3z + 5 \supset x \leqslant y + f(z')$.] Let F′ be the resulting formula.

(3) The negation of F′ is placed into a disjunctive normal form $G_1 \vee G_2 \vee \ldots \vee G_p$, as described in Section III-B-3. Each G_i is a conjunction of linear inequalities.

(4) The G_is are tested one by one for satisfiability by applying Steps (a), (b), and (c) below. If none is satisfiable, F is valid.

   (a) The ILP associated with G_i is solved, using either of the methods suggested in Section III-B-3.

   (b) If there is no solution, G_i is unsatisfiable.

   (c) Otherwise, the discovered solution is examined for violations of substitutivity of equality (in a way described momentarily). If there are no violations, G_i is satisfiable, and the discovered solution provides a counterexample for F. If, on the other hand, a violation is found, a formula H that summarizes the violated property of substitutivity is formulated. Step (4) is

now applied recursively to each of the conjunctions in the disjunctive expansion of $H \wedge G_i$. G_i is satisfiable (in the extended theory) if and only if each of these is satisfiable.

We have yet to show how to examine a given solution S for violations of substitutivity and how to generate the associated substitutivity formula. We must also show, of course, that the procedure [Step (4) in particular] terminates. For notational convenience, we assume in the explanation that follows that all uninterpreted function symbols appearing in F are monadic; the general case is a straightforward extension.

The technique for detecting violations if founded on the recursive function EQPAIRS defined below. The definition depends on the following conventions. Let S be the discovered integer solution for the G_i whose satisfiability is to be determined, and T the set of terms to which S assigns values. For each term $t \in T$, let S(t) designate the value assigned to t by S. Let U be the set of terms in T together with all of their subterms. [For example, if S is given by $x = 0$, $g(x) = 0$, and $g(g(g(g(x)))) = 1$, then $T = \{x, g(x), g(g(g(g(x))))\}$ and $U = \{x, g(x), g(g(g(g(x)))), g(g(x)), g(g(g(x)))\}$]. Finally, for each term $t \in U$, define the set:

$$E_t = \begin{cases} \{t' \in T \mid S(t) = S(t')\} & \text{if } t \in T \\ \{t\} & \text{if } t \notin T \end{cases}$$

EQPAIRS is defined as follows:

EQPAIRS $(t_1, t_2,$ alreadytried) =

if $\langle t_1, t_2 \rangle \in$ alreadytried, then return ϕ

else if $t_1 \in E_{t_2}$, then return $\{\langle t_1, t_2 \rangle\}$

else if for some function symbol f and terms u_1, u_2:

(i) $f(u_1) \in E_{t_1}$ and

(ii) $f(u_2) \in E_{t_2}$ and

(iii) EQPAIRS $(u_1, u_2,$ alreadytried $\cup \{\langle t_1, t_2 \rangle\}) \neq \phi$,

then return $P_1 \cup P_2 \cup P_3$, where

P_1 = if $t_1 = f(u_1)$ then ϕ else $\{\langle t_1, f(u_1)\rangle\}$

P_2 = if $t_2 = f(u_2)$ then ϕ else $\{\langle t_2, f(u_2)\rangle\}$

P_3 = EQPAIRS (u_1, u_2, alreadytried $\cup$ $\{\langle t_1, t_2\rangle\}$)

else return ϕ.

As is evident from the definition, EQPAIRS is a function of three arguments. The first two (t_1 and t_2) are bound to terms in U, and the third (alreadytried) is bound to a set of pairs in U x U. The third argument is always bound to the empty set on external calls, and comes into play on internal calls only as a device to prevent infinite recursion. EQPAIRS returns a set of pairs in T x T.

The usefulness of EQPAIRS turns on the following result, stated here without proof:

Theorem: If for all $t_1, t_2 \in T$, either $s(t_1) = s(t_2)$ or EQPAIRS $(t_1, t_2, \phi) = \phi$, then S has no violations of substitutivity, and hence G_i is satisfiable as a member of the extended theory. On the other hand, if for some t_1, t_2, $s(t_1) \neq s(t_2)$, and EQPAIRS $(t_1, t_2, \phi) = \{\langle r_1, s_1\rangle, \ldots, \langle r_n, s_n\rangle\}$, $n \geq 1$, then the formula

$$H \equiv [r_1 = s_1 \wedge r_2 = s_2 \wedge \ldots r_n = s_n \supset t_1 = t_2]$$

follows from substitutivity but is not satisfied by S.

To check S for violations of substitutivity, it thus suffices to compute EQPAIRS(t_1, t_2, ϕ) for pairs t_1, t_2 of terms in T assigned different values by S. A violation exists if and only if EQPAIRS $(t_1, t_2, \phi) \neq \phi$ for some such pair. In such a case, the formula H summarizes the violation.

Note from the definition of EQPAIRS that, if $t_1, t_2 \in T$, $s(t_1) \neq s(t_2)$, and EQPAIRS $(t_1, t_2, \phi) \neq \phi$ then t_1 and t_2 must be functional terms with the same outermost function symbol. In checking S for violations of substitutivity, therefore, one need only compute EQPAIRS (t_1, t_2, ϕ) for such pairs.

Note also that EQPAIRS is defined nondeterministically; there may be more than one choice of f_1, u_1, and u_2 that satisfies (i), (ii) and (iii) in the definition. It makes no difference which choice is made. Similarly, there may be several pairs t_1, t_2 for which EQPAIRS is nonempty. It suffices to generate H on the basis of the first such pair encountered.

Let us now return to the earlier example

$$F \equiv x \leqslant g(x) \wedge g(x) \leqslant x \supset x = g(g(g(g(x))))\quad .$$

Applying Steps (1), (2), (3) of the procedure,

$$G_1 \equiv x \leqslant g(x) \wedge g(x) \leqslant x \wedge x \leqslant g(g(g(g(x)))) - 1$$

and

$$G_2 \equiv x \leqslant g(x) \wedge g(x) \leqslant x \wedge g(g(g(g(x)))) \leqslant x - 1$$

are obtained as before. Solving G_1 once again produces the solution S: $x = 0$, $g(x) = 0$, $g(g(g(g(x)))) = 1$.

Since $g(x)$, $g(g(g(g(x))))$ form the only pair of terms in T with the same outermost function symbol and with different assigned values, it suffices to compute EQPAIRS for that pair only:

EQPAIRS $(g(x), g(g(g(g(x)))), \phi)$:

$E_{g(x)} = \{x, g(x)\}$

$E_{g(g(g(g(x))))} = \{g(g(g(g(x))))\}$

Letting $g(x)$ be $f(u_1)$, $g(g(g(g(x))))$ be $f(u_2)$, and recursing:

EQPAIRS $(x, g(g(g(x))), \{\langle g(x), g(g(g(g(x))))\rangle\})$:

$E_x = \{x, g(x)\}$

$E_{g(g(g(x)))} = \{g(g(g(x)))\}$

Letting $g(x)$ be $f(u_1)$, $g(g(g(x)))$ be $f(u_2)$:

EQPAIRS $(x, g(g(x)), \{\langle g(x), g(g(g(g(x))))\rangle, \langle x, g(g(g(x)))\rangle\})$:

$E_x = \{x, g(x)\}$

$E_{g(g(x))} = \{g(g(x))\}$

Letting $g(x)$ be $f(u_1$, $g(g(x))$ be $f(u_2)$:

EQPAIRS (x, g(x), {⟨g(z), g(g(g(g(x))))⟩, ⟨x, g(g(g(x)))⟩, ⟨x, g(g(x))⟩}):

$$E_x = \{x, g(x)\}$$

$$= \{\langle x, g(x)\rangle\}$$

$$= \{\langle x, g(x)\rangle\}$$

$$= \{\langle x, g(x)\rangle\}$$

$$= \{\langle x, g(x)\rangle\} \quad .$$

The formula H produced in this case is thus

$$x = g(x) \supset g(x) = g(g(g(g(x)))) \quad .$$

The ILPs corresponding to the conjunctions in the disjunctive expansion of $H \Delta G_1$ are all found infeasible; hence, G_1 is unsatisfiable.

In a similar manner, G_2 is found unsatisfiable. The procedure thus halts, reporting that F is valid.

In this last example, only one level of recursion in Step (4) was needed. In rare instances, more than one is required. Consider, for example, the theorem:

$$F \equiv a \leqslant b \leqslant f(a) \leqslant 1 \supset a + b \leqslant 1 \vee b + f(b) \leqslant 1 \vee f(f(b)) \leqslant f(a) \quad .$$

Applying Steps (1), (2), and (3) of the procedure, one obtains the single conjunction G:

$$a \leqslant b \wedge b \leqslant f(a) \wedge f(a) \leqslant 1 \wedge 2 \leqslant a + b \wedge 2 \leqslant b + f(b) \wedge f(a) + 1 \leqslant f(f(b))$$

Solving the corresponding ILP produces as one possibility the following solution S:

$$a = 1,\ b = 1,\ f(a) = 1,\ f(b) = 2,\ f(f(b)) = 2 \quad .$$

The only two pairs of terms for which EQPAIRS need be tried are f(a), f(b), and f(a), f(f(b)). Trying f(a), f(b) first, one immediately obtains a violation: EQPAIRS (f(a, f(b), ϕ) = {⟨a, b⟩}. The formula [a = b ⊃ f(a) = f(b)] ∧ G is now expanded into disjunctive form, giving the three conjunctions

$$a \leqslant b - 1 \wedge G,\ b \leqslant a - 1 \wedge G, \text{ and } f(a) = f(b) \wedge G \quad .$$

The ILPs associated with the first two of these are infeasible. The third one, however, yields a solution:

$$a = 1,\ b = 1,\ f(a) = 1,\ f(b) = 1,\ f(f(b)) = 2 \quad .$$

Applying EQPAIRS to the pair f(a), f(f(b)) produces $\{\langle a, f(b)\rangle\}$. Step (4) is again applied recursively to the three conjunctions in the expansion of $[a = f(b) \supset f(a) = f(f(b))] \wedge f(a) = f(b) \wedge G$. This time, all ILPs are found to be infeasible, and the procedure terminates.

6. Termination

It is easy to see, in fact, that the recursive Step (4) must always terminate. Otherwise, some branch of the computation would be infinite, yielding an infinite sequence of ILPs G, G′, G″ . . . with a corresponding sequence of solutions S, S′, S″ . . . and substitutivity formulas H, H′, H″ . . . . Since each solution assigns values to exactly the same set T of terms, some H must be repeated in the sequence. This is impossible, however, since each solution S fails to satisfy its corresponding H, but does satisfy all preceding Hs.

7. A More Efficient Version of the Procedure

Although the procedure given in Section III-B-5 dramatically improves on the naive reduction method, substantial additional improvement is possible.

First note that the expansion of $H \wedge G_i$ into disjunctive form in Step (4c) is unnecessary. The conjunctions that result from this expansion can be precomputed as follows:

$$\begin{aligned} H \wedge G_i &\equiv [r_1 = s_1 \wedge \ldots \wedge r_n = s_n \supset t_i = t_2] \wedge G_i \\ &\equiv [r_1 \neq s_1 \vee \ldots \vee r_n \neq s_n \vee t_1 = t_2] \wedge G_i \\ &\equiv r_1 \leqslant s_1 - 1 \wedge G_i \vee s_1 \leqslant r_1 - 1 \wedge G_i \\ &\quad \vdots \\ &\quad \vee\ r_n \leqslant s_n - 1 \wedge G_i \vee s_n \leqslant r_n - 1 \wedge G_i \\ &\quad \vee\ t_1 = t_2 \wedge G_i \quad . \end{aligned}$$

Note that $2n + 1$ conjunctions are thus generated, each one augmenting G_i with an inequality. If the ILP solver used can be operated incrementally (as can simplex-based methods), the new ILPs can be solved with little additional effort.

In the great majority of cases encountered in practice, further speedup is possible. Note from the definition of EQPAIRS that $S(r_j) = S(s_j)$ for each j, $1 \leqslant j \leqslant n$. Now suppose it can be established, for a given j, that r_j and s_j are equal in <u>all</u> solutions for G_i. In this case, the two conjunctions $r_j \leqslant s_j - 1 \wedge G_i$ and $s_j \leqslant r_j - 1 \wedge G_i$ are necessarily unsatisfiable and can therefore be dispensed with.

What makes this observation useful is that one can test whether r_j and s_j are equal in all solutions of G_i quite easily; it is necessary only to test (using the ILP solver) for $\max_{G_i} (r_j - s_j) = \min_{G_i} (r_j - s_j) = 0$. This can be done more quickly than testing $r_j \leqslant s_j - 1 \wedge G_i$ and $s_j \leqslant r_j - 1$ for feasibility, since it does not involve additional inequalities.

Returning to the earlier example:

$$G_1 \equiv x \leqslant g(x) \wedge g(x) \leqslant x \wedge x \leqslant g(g(g(g(x)))) - 1 \quad ,$$

$$H \equiv x = g(x) \supset g(x) = g(g(g(g(x))))$$

we see that x and g(x) must have equal values in all solutions of G_1 and so only the conjunction $g(x) = g(g(g(g(x)))) \wedge G_1$ needs to be tested.

These ideas suggest the following replacement for Step (4c):

(4c) The discovered solution S is tested for violations of substitutivity by computing EQPAIRS (t_1, t_2, ϕ) for pairs $t_1, t_2, \in T$ that have different values in S but the same outermost function symbol. If EQPAIRS returns ϕ for all such pairs, the solution S provides a counterexample for F and the procedure halts. If t_1, t_2 are found for which EQPAIRS $(t_1, t_2, \phi) = \{\langle r_1, s_1\rangle \ldots, \langle r_n, s_n\rangle\}$, $n \geqslant 1$, then Step (4) is applied recursively to

$$t_1 = t_2 \wedge G_i \quad ,$$

and for $1 \leqslant j \leqslant n$, to:

$$r_j \leqslant s_j - 1 \wedge G_i \text{ unless } \max_{G_i} (s_j - r_j) \leqslant 0$$

and $s_j \leqslant r_j - 1 \wedge G_i$ unless $\max_{G_i}(r_j - s_j) \leqslant 0$.

G_i is unsatisfiable if and only if each of the conjunctions thus tested is found unsatisfiable.

Finally, it might be remarked that the function EQPAIRS can be implemented much more efficiently than the definition suggests. If, for example, a table is used to record the results of internal calls, the amount of work required to compute EQPAIRS can be made to grow no faster than the square of the length of the input.

C. An Algorithm for Reasoning About Equality

1. Introduction

To be useful for program verification, a deductive system must be able to reason proficiently about equality. Important as its semantics are, equality is often handled in an ad hoc and incomplete way – most usually with a rewrite rule that substitutes equals for equals with some heuristic guidance. This section presents a simple algorithm for reasoning about equality that is fast, complete (for ground formulas with function symbols and equality), and useful in a variety of theorem-proving situations. A proof of the theorem on which the algorithm is based is given as well.

2. An Example

Let us first consider an example formula and how one could go about proving it. The formula given below is of the kind one encounters in verifying programs involving array indexing:

$$(I = J \wedge K = L \wedge A[I] = B[K] \wedge J = A[J] \wedge M = B[L]) \supset A[M] = B[K] \quad .$$

Here, A and B are function symbols (corresponding to arrays) while I, J, K, L, and M are universally quantified variables (corresponding to program variables).

One might approach such a formula by working backward from the conclusion, substituting equals for equals until the left-hand side is transformed into the right-hand side. With a little patience, the following proof is obtained:

	A[M]	
=	A[B[L]]	(using M = B[L])
=	A[B[K]]	(using K = L)
=	A[A[I]]	(using A[I] = B[K])
=	A[A[J]]	(using I = J)
=	A[J]	(using J = A[J])
=	A[I]	(using I = J again)
=	B[K]	(using A[I] = B[K] again) .

Of course, one could just as easily work from B[K] rather than from A[M], or work from both simultaneously; the links needed in the chain are the same in either case.

While this "backward substitution" method and other methods that transform formulas through a sequence of substitutions are logically sound, they are not particularly well-suited to machine deduction – simply because there is no easy way of knowing what substitution is the "right" one to make at each step. Indeed, a program working on the formula given above could grind on forever (for example, by repeated application of the substitution J → A[J]), generating terms of ever-increasing depth of nesting.

Intuitively, it would not seem necessary to generate terms beyond a certain depth. It is easy, however, to construct examples showing that the critical depth (the smallest depth necessary to consider) cannot be calculated solely as a function of the depths of the terms appearing in the original formula; in particular, the critical depth is not simply the maximum of these depths. The reader can easily convince himself, for example, that no backward-substitution proof can be carried out for the formula given above without generating a term of at least depth 3. (The maximum depth of terms occurring in the formula is only 2.) Even if one could conveniently calculate the critical depth, one would still, in general, generate many more terms than are necessary.

Fortunately, this difficulty with substitution transformation methods is not inherent in the problem. Section III-C-3 presents a more efficient method that considers only the terms appearing in the original formula.

3. The Procedure

The method given here may be described formally as a decision procedure for the subclass of predicate calculus with function symbols and equality whose formulas have

only universal quantifiers in prenex form. While the decidability of this subclass is well-known, the classical decision procedure for it [given by Ackermann (1954)] produces a combinational explosion that makes that method computationally infeasible for non-trivial problems.

Since the decision procedure presented here operates on the negation of the formula to be proved, it can also be viewed as a refutation procedure for ground formulas with function symbols and equality. The universally-quantified variables become Skolem constants in the Skolemization of the negation.

The procedure is as follows. The matrix of the formula F to be decided is first negated and placed in disjunctive normal form. Next, all atomic formulas other than equalities are replaced by equalities as follows. For each n-ary predicate symbol P_i^n occurring in the formula, a new n-ary function symbol f_i^n is introduced. Each atomic formula $P_i^n(t_1, t_2, \ldots t_n)$ occurring in the formula is then replaced by the equality $f_i^n(t_1, t_2 \ldots, t_n) = c$, where c is a constant. The modified d.n.f. is clearly intersatisfiable with the original one, and is satisfiable if and only if one of its disjuncts is satisfiable. Each disjunct, moreover, consists of a conjunction of equalities and negations of equalities. The problem is thus reduced to testing the satisfiability of each such conjunction.

For example, suppose the formula to be proved is:

$$[(P_z \vee x = z) \wedge P_x] \supset [P_{g(y)} \vee z \neq g(y)] \quad .$$

Putting the negation into disjunctive normal form, we have:

$$(P_z \wedge P_x \wedge \neg P_{g(y)} \wedge z = g(y)) \vee (x = z \wedge P_x \wedge \neg P_{g(y)} \wedge z = g(y)) \quad .$$

Introducing the new function symbol f to replace P, we obtain:

$$(f(z) = c \wedge f(x) = c \wedge f(g(y)) \neq c \wedge z = g(y))$$

$$\vee (x = z \wedge f(x) = c \wedge f(g(y)) \neq c \wedge z = g(y)) \quad .$$

It remains to show how to test each conjunction for satisfiability. Let S be the set of equalities and negations of equalities occurring in the conjunction to be tested. Let T be the set of terms and subterms occurring in S, and define the binary relation $\doteq$ as the

smallest relation over T × T (where $u_1, u_2 \ldots u_n, t_1, t_2, v_1, v_2 \ldots v_n$ denote terms and f denotes a function symbol) that:

(1) Contains all pairs $\langle t_1, t_2 \rangle$ for which $'t_1 = t_2' \in S$

(2) Is reflexive, symmetric, and transitive.

(3) Contains the pair $\langle f(u_1, u_2 \ldots u_r), f(v_1, v_2 \ldots v_r) \rangle$ whenever it contains the pairs $\langle u_i, v_i \rangle$, $1 \leqslant i \leqslant r$, and $f(u_1, u_2 \ldots u_r), f(v_1, v_2 \ldots v_r)$ are both in T.

The test for satisfiability of S depends on the following theorem (proved later):

Theorem: S is unsatisfiable ⇔ there exist terms $t_1, t_2 \in T$ such that

$$'t_1 \neq t_2' \in S \text{ and } t_1 \doteq t_2 \quad .$$

The theorem tells us that to determine the satisfiability of S it suffices to consider the negated equalities of S one at a time. If one is found (say $t_1 \neq t_2$) for which $t_1 \doteq t_2$, S is unsatisfiable; otherwise S is satisfiable. Note that the definition of $\doteq$ involves only terms in T.

In order to use the theorem, it is necessary to be able to calculate whether a given pair of terms is in the relation $\doteq$. This can be done in a straightforward way by building the relation from the definition (1) is used as a basis, and (2) and (3) are repeatedly applied until no new terms are generated. Since $\doteq$ is an equivalence relation, one can conveniently represent it during the construction as a collection of sets of elements of T, each set containing elements known to be in the relation with the other elements of that set.

As an illustration, consider the set S = {I = J, K = L, A[I] = B[K], J = A[J], M = B[L], A[M] ≠ B[K] } that arises from the example given earlier. The corresponding set T is {I,J,K,L, A[I], B[K], A[J], M, B[L], A[M] }. The relation $\doteq$ is constructed from its definition as follows.

From the basis (1), one obtains:

{{I,J}, {K,L}, {A[I], B[K] }, {J, A[J] }, {M, B(L)}}

Using (2):

{{I,J, A[J] } {K,L} {A[I], B[K] }, {M, B[L] }}

Using (3):

{{I,J, A[J] } {K,L} {A[I], B[K] }, {M, B[L] }, {A[I], A[J] }, {B[K], B[L] }}

Using (2):

{{I,J, A[J], A[I], B[K], B[L], M}, {K,L}}

Using (3):

{{I,J, A[J], A[I], B[K], B[L], M}, {K,L}, {A[M], A[I] }, {A[M], A[J] }}

Using (2):

{{I,J, G[J], A[I], B[K], B[L], A[M] }, {K,L}}

Since (3) yields no new pairs, the construction is complete. Since A[M] $\doteq$ B[K], S must be unsatisfiable.

The rules for building up $\doteq$ can be implemented quite efficiently. Oppen and Nelson (1977) have recently coded a very fast implementation that represents terms as graphs and uses the Tarjian (1975) set-union algorithm in the closure step. Oppen and Nelson have shown that their implementation requires only order n^2 deterministic time and linear space, where n is the length of the input S.

While the satisfiability of each set S can thus be determined quite quickly, the procedure as a whole (and the expansion into disjunctive normal form in particular) is of exponential time complexity. This is not surprising, of course, since the decision problem for the class is NP-complete.

The author has coded the procedure in INTERLISP for the PDP-10 using a matrix representation of $\doteq$. The program has been tested on a few dozen examples of the kind that arise in program verification applications. It was found that most examples four or five lines long could be handled in just a few seconds. The example presented at the beginning of the paper required less than a second.

4. Proof of the Theorem

The main import of the theorem on which the algorithm is based is that it suffices to "consider" only the terms occurring in formula to be decided. The proof of the theorem is largely concerned with extending the model provided by the relation $\doteq$ from

the finite set T to the entire Herbrand universe. We now restate the theorem and give its proof.

Theorem:

S is satisfiable $\Leftrightarrow$ there exist no $t_1, t_2 \in T$ such that

$$t_1 \doteq t_2 \text{ and } 't_1 \neq t'_2 \in S.$$

Proof:

$\Rightarrow$ Suppose S is satisfiable, $t_1, t_2 \in T$ and $t_1 \doteq t_2$. Let M be a model for S. Because M satisfies the reflexivity, symmetry, transitivity, and substitutivity axioms of equality, $t_1 \doteq t_2$ implies that t_1 and t_2 must have the same values in M. Hence, $'t_1 \neq t'_2$ cannot be a member of S.

$\Leftarrow$ Suppose there are no terms t_1, t_2 in T such that $t_1 \doteq t_2$ and $'t_1 \neq t'_2 \in S$. We will show that S is satisfiable by constructing a model M for S. The model must assign a value $v_M(t)$ to each term t in the Herbrand universe of S in such a way that:

(1) $'t_1 = t'_2 \in S$ implies $v_M(t_1) = v_M(t_2)$

(2) $'t_1 \neq t'_2 \in S$ implies $v_M(t_1) \neq v_M(t_2)$

(3) $v_M(x_i) = v_M(y_i)$, $1 \leqslant i \leqslant r$, implies $v_M(f(x_1, \ldots x_r)) = v_M(f(y_1, \ldots, y_r))$
(where f ranges over all function symbols and x_i, y_i over all terms)

The first two conditions require that M satisfies each atomic formula of S. The third condition requires M to satisfy the substitutivity axiom of equality.

Before defining v_M we first construct the term universe

$T_\infty = \bigcup_{i=o}^{\infty} T_i$ of S inductively as follows:

$T_o = T$

$T_{i+1} = \{f(t_1, \ldots, t_r) \mid t_i \in T_i\} \cup T_i$

(where f ranges over all function symbols occurring in S)

Note that the term universe T_∞ is identical as a set to the Herbrand universe, but is constructed differently.

Next, pick a representative term from each of the equivalence classes induced by $\doteq$ on T, and define the function $a : T \to T$ that assigns to each term in T the representative of its class.

The model M is now constructed inductively as follows:

I. If $t \in T_o$, let $v_M(t) = a(t)$

II. If $t \in T_{j+1} - T_j$, $j \geqslant o$, and $t = f(t_1, t_2, \ldots t_r)$, then let

$$v_M(t) = \begin{cases} v_M(f(x_1, \ldots, x_r)) \text{ if there exists no } f(x_1, \ldots, x_r) \in T_j \\ \qquad\qquad \text{and } v_{M(x_i)} = v_M(t_i),\ 1 \leqslant i < r \\ f(v_M(t_1), \ldots, v_M(t_r)) \quad \text{otherwise} \end{cases}$$

Note that M is a Herbrand model, i.e., it always assigns values from the Herbrand universe. The notation "$f(v_M(t_1), \ldots, v_M(t_r))$" is intended to represent the function symbol denoted by f followed by the terms obtained by evaluating $v_M(t_i)$ for each i.

Note also that v_M would not seem to be uniquely defined, owing to the existential choice implicit in the definition. In a moment, however, it will be clear that only one choice is possible.

Now, we need to show that M satisfies (1), (2), and (3) above. (1) and (2) hold since $'t_1 = t'_2 \in S \Rightarrow t_1 \doteq t_2 \Rightarrow a(t_1) = a(t_2) \Rightarrow v_M(t_1) = v_M(t_2)$ and

$$'t_1 \neq t'_2 \in S \Rightarrow t_1 \doteq t_2 \Rightarrow a(t_1) \neq a(t_2) \Rightarrow v_M(t_1) \neq v_M(t_2)$$

It remains to show that (3) holds, i.e., that $v_M(x_i) = v_M(y_i)$, $1 \leqslant i \leqslant r$, implies that $v_M(f(x_1, \ldots, x_r)) = v_M(f(y_1, \ldots y_r))$. This is proved by induction on the maximum m of the term universe heights of $f(x_1, \ldots, x_r)$, $f(y_1, \ldots, y_r)$:

Basis: $m = o$

Then x_i, y_i, $f(x_1, \ldots, x_r)$, $f(y_1, \ldots, y_r)$ are all in T, and so

$v_M(x_i) = v_M(y_i) \Rightarrow a(x_i) = a(y_i) \Rightarrow x_i \doteq y_i$

$\Rightarrow f(x_1, \ldots, x_r) \doteq f(y_1, \ldots, y_r) \Rightarrow a(f(x_1, \ldots x_r)) = a(f(y_1, \ldots y_r))$

$\Rightarrow v_M(f(x_1, \ldots, x_r)) = v_M(f(y_1, \ldots, y_r))$ as required

Induction Step: $m > o$.

First consider the case in which the height of $f(x_1, \ldots, x_r)$ is strictly greater than that of $f(y_1, \ldots, y_r)$. In this case, $v_M(f(x_1, \ldots, x_r)) = v_M(f(z_1, \ldots, z_r))$, where $f(z_1, \ldots, z_r)$ is of lesser height than $f(x_1, \ldots, x_r)$, and $v_M(x_i) = v_M(z_i)$. (Note that $f(z_1, \ldots, z_r)$ is possibly the same as $f(y_1, \ldots, y_r)$.) Now since $v_M(y_i) = v_M(x_i) = v_M(z_i)$, we have by induction hypothesis that $v_M(f(x_1, \ldots, x_r)) = v_M(f(z_1, \ldots, z_r)) = v_M(f(y_1, \ldots, y_r))$ as required.

In the remaining case, $f(x_1, \ldots, x_r)$ and $f(y_1, \ldots, y_r)$ are of the same height. Now if there exists a term $f(z_1, \ldots, z_r)$ of lower height such that $v_M(z_i) = v_M(x_i)$, the argument above can be used. Otherwise, $v_M(f(x_1, \ldots x_r)) = f(v_M(x_1), \ldots, v_M(x_r)) = f(v_M(y_1), \ldots, v_M(y_r)) = v_M(f(y_1, \ldots, y_r))$ as required.

Q.E.D.

It might be noted that (3) implies the uniqueness of M as it has been defined above. Of course, the uniqueness was not essential to the proof.

IV CONCLUSIONS

A. Introduction

In this final section we present conclusions both with respect to the state of the program verification art and to specific project goals.

The needs that motivated research on program verification are no less valid today than they were when we began work on the project four years ago. If anything, the desirability of producing and maintaining reliable software has increased during this period of time.

While there is no question as to the potential payoff of formal proof (and related techniques, such as methodologies for formal specification) in achieving reliable software, this benefit must be assessed in relation to cost. At the current state of the art, the cost of verifying all but the most insubstantial programs is quite high, both in terms of the sophistication required of the user of the verification techniques, and the amount of time and effort required to carry the verification process through to completion. The current state of the art of program verification is analogous to that of the development of nuclear fusion technology--the potential benefits are vast, but the break-even point is still ahead of us. In

both cases, however, the difficulties that stand in the way of success are engineering problems as opposed to insurmountable theoretical difficulties.

While the practical use of verification technology outside the research laboratory is still a thing of the future, the techniques that have been developed here and elsewhere have already had tangible benefits. As a conceptual tool for guiding specification, design, and even implementation, the notion of proof of correctness has had considerable impact. Manual proofs of correctness (semiformal ones, for the most part) are beginning to be employed in design of large systems (e.g., PSOS, SIFT). The impact has also been felt strongly in the area of language design. Verification considerations have played a role in the design of several new languages (notably PASCAL and EUCLID), and they will probably assist in the design of such still newer languages as the common high-order language for DoD use. The formalisms of Floyd, and even more strongly those of Hoare (proof rules and axioms) and Dijkstra (predicate transformers) have altered the way in which some language users and designers think about programs and programming languages. We foresee this trend continuing, perhaps even accelerating, in the next few years. Moreover, as the difficulties of automatic verification are gradually

overcome, program verification systems will eventually become commonplace tools in sophisticated software development environments. The RADC system of which our current effort is in support will (we hope) be among the first of these.

B. Difficulties of Verification

The difficulties of formal proof of correctness can be divided into four categories:

* Problems of specification
* Idealization of real machine environments
* Formulation of inductive assertions (in Floyd's Method)
* Theorem-proving.

Problems associated with specification are inherent in any formal approach to software validation. While the development of powerful assertion languages can make the job of specification easier (just as high-level languages have made the job of programming easier), there can be no substitute for thinking clearly about just what a program is supposed to accomplish. In the final analysis, any solution to the problem of formal specification will depend in good part on programmer training. This will require considerable research and experience in practical environments, and will go hand in hand with the adoption of good structuring practices.

The problem of idealization of real machine environments arises from discrepancies between the program semantics assumed by the verification system and those that are actually operative in the machine environment in which the program is run. While this problem is often cited by critics of verification, it is actually less a problem of verification than it is a problem of mismatch between the programming language and its implementation. Unless the semantic specification of a programming language is met by that language's implementation, reliable programs are an impossibility, regardless of the reliability technology used.

The third and fourth difficulties cited above are the primary ones addressed by our four-year effort. As we have already described the results of this work in the body of this report, we give only a summary account here.

C. Results of The Four-Year Effort

The first two years of the project focused almost exclusively on the development of semiautomatic aids for the invention of loop assertions. This work resulted in useful, but limited, techniques for numerically oriented programs. As the research progressed, however, it became increasingly clear that no single such technique could handle enough program structures to be of practical significance. One of

the greatest benefits of this phase of the work was that it spurred us to challenge the conceptual separation of the formulation of inductive assertions from the theorem-proving aspect of the verification process. We became increasingly aware of the similarity of intellectual activity required to perform these tasks successfully. In particular, it became clear that the formulation of loop assertions was very much akin to the generalization of inductive hypotheses in inductive styles of theorem proving (as, e.g., in the work of Boyer and Moore).

Accordingly, the last few years of the project have been concerned with deduction and with the exploration of the relation between Floyd's technique and a number of alternative methods of inductive proof. These include the transformation of programs into primitive recursive form before verification, the method of generator induction, the use of hierarchical structuring techniques, and methods related to subgoal and computation induction. These last two approaches were analyzed in detail and compared with Floyd's method to arrive at a unified understanding of their mutual relationships. In addition, two theorem-proving algorithms were developed in connection with both verification condition proof and interactive debugging of loop assertions. Both algorithms have been coded in LISP

and successfully used in support of an experimental program verifier for JOVIAL programs under a separate RADC contract.

REFERENCES

Ackerman, W., Solvable Cases of the Decision Problem, pp. 102-103 (North Holland Publishing Company, Amsterdam, 1954).

Basu, S. K., and J. Misra, "Proving Loop Programs," IEEE Trans. Software Eng., Vol. SE-1, No. 1, pp. 76-86 (March 1975).

Bledsoe, W. W., "The Sup-Inf Method in Presburger Arithmetic," Memo ATP-18, Mathematics Department, University of Texas, Austin, Texas (December 1974).

Bledsoe, W. W., "A New Method for Proving Certain Presburger Formulas," Advance Papers of the 4th International Joint Conference on Artificial Intelligence, Tibilisi, Georgia, USSR, p. 15 (September 1975).

Boyer, R. S., and J S. Moore, "Proving Theorems about Lisp Functions," J. ACM., Vol. 22, No. 1, pp. 129-144 (January 1975).

Boyer, R. S., J S. Moore, and R. E. Shostak, "Primitive Recursive Program Transformations," Proc. 3rd ACM Symposium on Principles of Programming Languages, Atlanta, Georgia, pp. 171-174 (January 1976).

Burstall, R. M., "Proving Properties of Programs by Structural Induction," Computer J., Vol. 12, No. 1, pp. 41-48 (February 1969).

Cooper, D. C., "Theorem Proving in Arithmetic Without Multiplication," in Machine Intelligence, Vol. 7, pp. 91-99 (American Elsevier, New York, 1972).

Downey, P., "Undecidability of Presburger Arithmetic with a Single Monadic Predicate Letter," Technical Report 18-72, Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts (1972).

Deutsch, L. P., "An Interactive Program Verifier," Ph.D. Dissertation, Department of Computer Science, University of California, Berkeley, California (1973).

Elspas, B., "The Semiautomatic Generation of Inductive Assertions for Proving Program Correctness," Interim Report, Contract F44620-73-C-0068, SRI Project 2686, Stanford Research Institute, Menlo Park, California (July 1974).

Elspas, B., R. S. Boyer, R. E. Shostak, and J. M. Spitzen, "A Verification System for JOVIAL/J3 Programs (Rugged Programming Environment--RPE/1)," Final Technical Report RADC-TR-76-58, Rome Air Development Center, New York [available through NTIS as AD-A024 395/6WC] (March 1976).

Elspas, B., R. E. Shostak, and J. M. Spitzen, "A Verification System for JOCIT/J3 Programs (Rugged Programming Environment--RPE/2)," Final Technical Report RADC-TR-77-229, Rome Air Development Center, New York [available through NTIS] (June 1977).

Floyd, R. W., "Assigning Meanings to Programs," in Mathematical Aspects of Computer Science, J. T. Schwartz, ed., pp. 19-32 (American Mathematical Society, Providence, Rhode Island, 1967).

German, S. M., and B. Wegbreit, "A Synthesizer of Inductive Assertions," IEEE Trans. Software Eng., Vol. SE-1, No. 1, pp. 68-75 (March 1975).

Gomory, R. E., "An Algorithm for Integer Solutions to Linear Programs," in Recent Advances in Mathematical Programming, R. L. Graves and P. Wolfe, eds., pp. 269-302 (McGraw-Hill Book Company, New York, 1963); originally Princeton IBM Math. Research Report (November 1958).

Kreisel, G., and J. L. Krevine, Elements of Mathematical Logic, pp. 54-57 (North Holland Publishing Company, Amsterdam, 1967).

Lee, R. D., "An Application of Mathematical Logic to the Integer Linear Programming Problem," Notre Dame J. Formal Logic, Vol. 13, No. 2 (April 1972).

McCarthy, J., "Towards a Mathematical Science of Computation," Proc. IFIP Congress 1962, pp. 21-28 (North Holland Publishing Company, Amsterdam, 1962).

Manna, Z., and A. Pnueli, "Formalization of Properties of Recursively Defined Functions," Proc. ACM Symposium on Theory of Computing, pp. 201-210 (Association for Computing Machinery, New York, 1969).

Manna, Z., and A. Pnueli, "Formalization of Properties of Functional Programs," J. ACM, Vol. 17, No. 3, pp. 555-569 (July 1970).

Mendelsohn, E., Introduction to Mathematical Logic (D. Van Nostrand Co., Inc., Princeton, New Jersey, 1964).

Moore, J S., "Introducing Iteration into the Pure LISP Theorem Prover," IEEE Trans. Software Eng., Vol. SE-1, No. 3, pp. 328-338 (September 1975).

Moriconi, M., "Semiautomatic Synthesis of Inductive Predicates," Report ATP-16, Departments of Mathematics and Computer Science, University of Texas, Austin, Texas (June 1974).

Morris, J. H., and B. Wegbreit, "Subgoal Induction," Comm. ACM, Vol. 20, No. 4, pp. 209-222 (April 1977).

Oppen, D., Ph.D. Thesis, University of Toronto [article to appear in JCSS] (1975).

Oppen, D., and G. Nelson, "Fast Decision Algorithms Based on Union and Find," Proc. 8th Symposium on Foundations of Computer Science, Princeton, New Jersey (November 1977).

Presburger, M., "Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen in welchem die Addition als einzige Operation hervortritt," pp. 92-101 in Sprawozdanie z I Kongresu Matematykow Krajow Slowiańskich, Warszawa (1929).

Robinson, G., and L. Wos, "Paramodulation and Theorem-Proving in First-Order Theories with Equality," in Machine Intelligence, Vol. 4 D. Michie and B. Meltzer, eds., pp. 135-150 (American Elsevier, New York, 1969).

Robinson, L., and K. N. Levitt, "Proof Techniques for Hierarchically Structured Programs," Comm. ACM, Vol. 20, No. 4, pp. 271-283 (April 1977).

Shostak, R. E., "An Algorithm for Reasoning about Equality" [reproduced as Section III-C of this report], Proc. 5th Annual Conf. on Artificial Intelligence (IJCAI-77), Vol. 1, pp. 526-527 (August 1977a).

Shostak, R. E., "An Efficient Decision Procedure for Arithmetic with Function Symbols" [reproduced as Section III-B of this report], submitted for publication (1977b).

Shostak, R. E. "On the Sup-Inf Method for Proving Presburger Formulas," J. ACM, Vol. 24, No. 4, pp. 529-543 (October 1977c).

Spitzen, J. M., K. N. Levitt, and L. Robinson, "An Example of Hierarchical Design and Proof," Tech. Report 2, Contract N00014-75-C-0816, SRI Project 4079, Stanford Research Institute, Menlo Park, California (March 1976) [Revised version submitted for publication (August 1977)].

Suzuki, N., "Verifying Programs by Algebraic and Logical Reduction," Proc. Int'l. Conf. Reliable Software, Sigplan Notices, Vol. 10, No. 6 (June 1975).

Tarjian, R., "Efficiency of a Good but Not Linear Set-Union Algorithm," J. ACM, Vol. 22, No. 2, pp. 215-225 (April 1975).

Wegbreit, B., and J. M. Spitzen, "Proving Properties of Complex Data Structures," J. ACM, Vol. 23, No. 2, pp. 389-396 (April 1976).

Appendix A

AN EXAMPLE OF HIERARCHICAL DESIGN AND PROOF

Jay M. Spitzen
Karl N. Levitt
Lawrence Robinson

Revised: August 1977

CONTENTS

ILLUSTRATIONS

I INTRODUCTION*

The use of structuring techniques in programming, for example programming by successive refinement [5] (also called hierarchical programming), has been recognized as increasingly helpful in the design and management of large system efforts. A number of such design techniques are now promoted for routine use in commercial software development [33]. Some of these techniques are also alleged to permit the verification of large systems by reducing them to a collection of small programs, each easily verified.

Important questions about such hierarchical proofs are:

* Can systems be decomposed into subprograms that can be characterized by clear and natural assertions?
* Can proofs of the subprograms be combined to demonstrate the correctness of the system?
* Is it generally possible to formulate and prove significant implementation-independent properties of systems?

Several recent developments yield positive answers: these are the hierarchical design and module specification techniques of [24] and the data abstraction techniques of [9] and [26]. (The word "module" is very widely and imprecisely used and the reader should be wary of drawing inferences not based on our very specific use.) A module is the basic unit in a hierarchical decomposition--a collection of operations and data. The module permits the definition of complex abstract types. For example, a type "file" can be defined by a module with operations for creating a file, inserting a record into a file, reading a record, appending two files, etc., and data structures recording the file's contents.

---

* We thank B. Elspas, R. Boyer, and the referees for their helpful suggestions; R. Boyer and J Moore for their work on formalizing SPECIAL; and S. German for the idea of trying to prove Theorem 3. The research described has been supported by the Office of Naval Research (Contract N00014-75-C-0816), the National Science Foundation (Grant DCR74-18661), and the Air Force Office of Scientific Research (Contract F44620-73-C-0068).

To permit hierarchical proof, one must formally specify the modules of a hierarchical system. Styles of specification and their mathematical foundations differ (e.g., consult [8], [24], [27]; also, [15] and [13] are overviews), but the basic aim is to achieve abstract specification, i.e., specification that describes the input-output behavior of a module without recourse to an implementation of the module. This may be done in terms of the sequences of operations that have been been performed on the module, or by abstracting from these sequences to a module state. In the first of these approaches, one may attempt to describe concisely the infinite class of possible histories by a small number of "algebraic specifications", as in [8]. In this paper, we will use the state approach. An important aspect of a good specification--in any method--is that, for a properly conceived module, it is a concise, intuitive, and precise characterization of the behavior of the module, successfully abstracting from the details of any implementation of module. It is often possible to formulate and prove important properties of a module in terms of its specifications.

Similarly, both algebraic and state styles of specification lend themselves to two-stage implementations: First the data structures of a module (other than the most primitive) are represented in terms of lower level data structures (as in [9] and [26]) and, second, abstract programs are written for each operation in terms of lower level operations. Each abstract program may then be proved to satisfy its specifications on the assumption that the more primitive modules it employs are correct, given the specific data representation used.

Should descriptions of hierarchical structure, formal specifications of modules, and implementations all be written in a single language? We have chosen to separate these functions, and will describe a powerful specification language, SPECIAL, and a very simple implementation language ILPL. An alternative is to provide abstraction directly in the implementation language--the approach of [4], [14], [32], [11], and [31]. A second language issue is what characteristics a language (or, in our view, set of languages) should have to enable the

correct implementation of hierarchies of abstractions. For example, what protection principles are needed to ensure the data integrity of a module?

The primary contributions of this paper are a description of the design and proof of a non-trivial, useful program, and a demonstration of a technique that has promise of making proof and formal description possible for large programs. Our example is a program to maintain unique lists with an efficient underlying implementation. We have attempted to address three classes of readers: those who wish to learn about formal specification should be able to do so by following our specifications and the associated prose; those who wish to learn about so-called "functional program verification" will be introduced this style of proof; and those who are unfamiliar with list processing may obtain an introduction to some relevant techniques.

In the next section, we introduce our design and proof method. Next, in Section III and Section IV, we present formal specifications of the two modules that comprise the top-level machine of an illustrative hierarchy. Based on these specifications, we are then able, in Section V, to prove several properties of this machine. The implementation of this machine and a proof of the correctness of this implementation are presented in Section VI and Section VII. Next, Section VIII outlines how the hierarchy described up to that point might be refined into an executable program. Finally, Section IX presents some concluding remarks.

II DESIGN AND PROOF METHOD

Suppose a programming problem P and a machine M are given and it is required to construct a program C that executes on M to solve P. M may be either a physical machine or a virtual machine provided, for example, by the compiler for a particular programming language. We believe that the program construction should proceed as follows.

First, we design an abstract computer AM on which it is easy to solve P. AM is designed by describing its states and executable instructions. We deliberately leave vague the meaning of an "easy solution" of P--for some purposes this will be a solution by a program AC that is only a page or two long; for other purposes it will be a solution by a program that can be mechanically proved correct using state-of-the-art verification systems. Having designed AM, we implement a solution of P on AM, and, in practice, usually alter the design of AM in the process.

If AM is the same as M, then we have satisfied the original requirement if it can be demonstrated that the solution is correct. Otherwise, we have reduced the original requirement to the new one of realizing AM in terms of M. To do this, we design a new machine AM' on which it is easy to realize AM. We represent the states of the first machine AM in terms of the states of AM' and we implement the executable instructions of AM by means of a set of programs AC' on AM'. The choice of <u>representation</u> or <u>implementation</u> may prove awkward; if so, we resort to altering the design of AM' or even that of AM. (Unfortunately, we usually cannot alter the original requirement P, though requirements sometimes are changed when a problem is better understood.)

Next, the realization of AM on AM' must be verified. This verification means that the (partial) solution of P obtained by executing AC on AM is equivalent to the (more complete) solution obtained by executing AC and AC' together on AM'. In this latter solution, the execution of an instruction of AM by AC is viewed not as

primitive but as a call on the subroutine in AC' that realizes that instruction on AM'. Again, if AM' is the same as M, we are done and otherwise we must continue to approach M by extending the sequence of machines AM, AM', ... and programs AC, AC', ... When we have extended this sequence to a program that executes on M, then the set of programs and subroutines AC, AC', ... constitutes the required the required solution C on machine M.

This description of programming has been phrased in the top-down paradigm, but that is not what is important. To make the programming of large problems feasible, reliable, and controllable, they must be somehow divided into small parts. We have no special preference for top-down or bottom-up programming in arriving at this division, and suspect that a flexible mixture of both techniques is required in general. We do advocate the use of formal methods to describe the division, to validate the resulting design, and to prove the correctness of the final program.

The product of our endeavors will thus consist of:

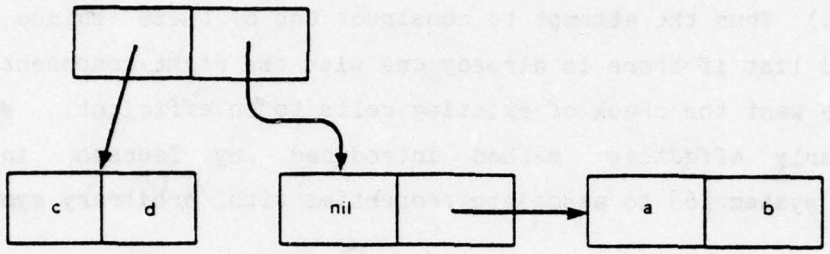
* A hierarchy of abstract machines
* A formal specification of each machine
* A representation of the states of each machine (except the given machine) in terms of the states of the machine below it
* An implementation of the instructions of each machine (except the given machine) in terms of the instructions of the machine below it

We specify an abstract machine using a method originally proposed by Parnas [24] and subsequently extended by Robinson and Levitt [26]. (Our formal specification language, SPECIAL, is described in [28].) A machine has a state and an instruction set. We give the state by describing the initial values of a set of V-functions. We give the instructions, called OV-functions, by describing how each changes the state of the machine and what value it returns. (The return of a particular value may, for formal purposes, be thought of as part of the change of state.)

A V-function specification consists of a header that describes its arguments and result and an initialization that describes the values of the function in an initial machine state.

An OV-function specification consists of a header that describes its argument structure, an assertion list stating preconditions on the calls of the function, and an exception list describing when its execution may have no effect other than signalling an error, and a set of effects that non-procedurally describe the changed state due to an execution of the function by defining the resulting values of the V-functions of the machine. (These values are described in terms of the old values of the V-functions and the arguments to the OV-function.) The effects include, if appropriate, the designation of the value to be computed and may allow a nondeterministic choice of successor state.

It is usually possible to give additional structure to an abstract machine M by describing it as the "product" of modules M1, M2, ..., Mn. When we do this, we will refer to the Mi as submachines or modules of M; otherwise the terms machine and module are used as synonyms. To form such a product, we require that the functions of the Mi be renamed to avoid conflicts. M has as its V-functions each of the V-functions of the Mi with the same initial sections. M has as its OV-functions each of the OV-functions of the Mi, with augmented effects sections. Specifically, if I is an OV-function of Mi and V is a V-function of Mj, where i~=j, we add to the effects of I the assertion that V is not changed by the execution of I.



(a) A PICTURE OF THE LIST ((c . d) nil a . b)

	CAR	CDR
100	101	103
101	c	d
102	101	105
103	nil	104
104	a	b
105	nil	104

(b) TWO DISTINCT ISOMORPHIC VERSIONS OF THE LIST ((c . d) nil a . b) AT 100 AND 102.

Figure 1. Distinct Isomorphic Lists

III THE ULIST MODULE

In this section and the next section, we present an abstract machine consisting of two modules: one that maintains conventional list structures and one that maintains a class of unique lists--lists such that no two are structurally isomorphic. (Figure 1 illustrates the usual realization of conventional lists where distinct isomorphic lists are possible.) Thus the attempt to construct one of these unique lists yields an old list if there is already one with the right components. * Naturally, we want the check of existing cells to be efficient. We use a particularly effective method introduced by Deutsch in his verification system [6] to associate properties with arbitrary symbolic expressions.

We begin by presenting a formal specification of a machine providing unique lists, and explain our notation by referring to this specification. The state of a ULIST machine is determined by what unique list cells exist. Hence we want a single V-function, UCELL(X1,X2) whose value is the cell with X1 and X2 as components--if there is any such cell--and the distinguished value "?" if there is no such cell. (All that will matter about "?" in this paper is that it does not satisfy the predicate ATOMP to be introduced in Section III.) There are four instructions on the ULIST machine: UCONS to obtain a list with specified components, UCAR and UCDR to extract the components of a ULIST list, and ULISTP to test whether an arbitrary object is a ULIST list. The specification is given in Figure 2.

It will be common, in the specification of this module, to ask the question, "Is the object X a unique list cell?" Therefore we introduce

---

* As an example of the utility of such a facility, suppose we save the property SIMPLIFIES-TO-ZERO in some table under--as key--the address of the list (SUBTRACT x x). If we subsequently independently create a conventional list of the same form, it will have a different address and the property will not be retrieved. But if both are unique then their addresses will be the same so that the property can be looked up successfully.

```
module:   ULIST:
  forall: Z1,Z2

  vfns:   UCELL(X1,X2)
           initial UCELL(X1,X2) = ?

  define: ISUCELL(X) = (exists X1,X2 : UCELL(X1,X2)=X~=?)

  ovfns:  UCONS(X1,X2) -> X
           assert  ISUCELL(X1) or ATOMP(X1)
                   ISUCELL(X2) or ATOMP(X2)
           effects if UCELL(X1,X2) = ?
                      then UCELL(Z1,Z2)~=X and
                           'UCELL(X1,X2)=X and
                           X~=? and ~ATOMP(X) and
                           (X1~=Z1 or X2~=Z2
                                   => 'UCELL(Z1,Z2)=UCELL(Z1,Z2))
                      else 'UCELL(Z1,Z2)=UCELL(Z1,Z2) and
                           X = UCELL(X1,X2)
          UCAR(X) -> X1
           assert  ISUCELL(X)
           effects 'UCELL(Z1,Z2)=UCELL(Z1,Z2)
                   UCELL(Z1,Z2)=X => X1=Z1
          UCDR(X) -> X2
           assert  ISUCELL(X)
           effects 'UCELL(Z1,Z2)=UCELL(Z1,Z2)
                   UCELL(Z1,Z2)=X => X2=Z2
          UCONSP(X) -> B
           effects 'UCELL(Z1,Z2)=UCELL(Z1,Z2)
                   B = ISUCELL(X)
```

Figure 2. Specification of ULIST Module

the abbreviation ISUCELL, expressing this predicate in terms of the module's single V-function. Note also that in the specification we use the predicate ATOMP to distinguish the objects that may be the "leaves" of list structures from the objects that are list cells. Rather than implementing this predicate as we develop our hierarchy, we will simply assume that it is present in the most primitive machine of the hierarchy and is reflected upward, with the same meaning, in each nonprimitive machine. In Section IX, we will discuss the significance of this assumption.

In the initial ULIST state, there are to be no list cells. We specify this by requiring that UCELL(Z1,Z2) be "?" initially. (We abbreviate slightly by listing at the head of each module symbols that should be read as universally quantified in all their uses in the module specification; for ULIST these are Z1 and Z2.)

Next, we specify the instruction UCONS(X1,X2) to obtain a cell with components X1 and X2. This instruction has no exceptions: it is required to achieve its effects for any arguments and state; in particular, this requires that any implementation have an unlimited set of cells. (Although this requirement is idealistic, it simplifies our presentation; SPECIAL does provide for the description of resource errors.) We assert that the arguments to UCONS are either outputs of UCONS [ISUCELL(X)] or atoms [ATOMP(X)]. Its effects are stated with an "if-then-else" assertion. We need to refer to two sets of values of UCELL--those associated with the state before the UCONS instruction is executed and those reflecting the changed state due to the execution. We will do this by writing UCELL(X1,X2) to refer to the old state and 'UCELL(X1,X2) to refer to the changed state. First, if the machine state is such that UCELL(X1,X2) is "?", then a new cell must be created. We do not choose to specify how cells are represented (e.g., by their integer addresses on some machine), but say only that the new cell is a value X that is not a cell before the execution of this instruction (i.e., UCELL(Z1,Z2)~=X for any X1, X2) and is the cell with the specified components afterwards ['UCELL(X1,X2)=X]. Besides constraining the value of the new cell, we must ensure that no other cells are affected by the instruction. Thus we say that if (Z1,Z2) is any pair of cell components other than the (X1,X2) given in the instruction call, then the new cell associated with (Z1,Z2) is the same as the old ['UCELL(Z1,Z2)=UCELL(Z1,Z2)].

If UCELL(X1,X2) is not "?", then the effects of the instruction are simpler. We constrain the result of UCONS to be the existing cell [Z=UCELL(X1,X2)] and require the new state to have exactly the same cells as the old ['UCELL(Z1,Z2)=UCELL(Z1,Z2)].

Next, we specify the UCAR and UCDR instructions. Our notion of the ULIST module is that it is not meaningful to ask for either of the components of an object other than a ULIST cell. Hence, we assert that the arguments to UCAR and UCDR are ULIST cells using the predicate ISUCELL which requires that its argument be in the image, under UCELL, of the set of pairs (Z1,Z2). The effects of UCAR and UCDR are similar. If Z1 and Z2 are such that UCELL(Z1,Z2) is the argument to UCAR or UCDR, then the UCAR component of UCELL(Z1,Z2) is Z1 and the UCDR component is Z2. (It is not obvious that such a specification is noncontradictory; this is a consequence of the theorem, proved below, that UCELL is single-valued: it maps distinct arguments to the same result only when that result is "?".) Besides giving the values of these functions, the specification asserts that they have no effect on UCELL ['UCELL(Z1,Z2)=UCELL(Z1,Z2)].

The last ULIST instruction is UCONSP. It is like UCAR and UCDR in that UCELL is unchanged. Its result B must be true or false, and true if and only if its argument is a ULIST list. But this is easily stated in terms of the UCELL V-function--it is equivalent to the existence of a pair (Z1,Z2) such that UCELL(Z1,Z2) is equal to the argument X [B = exists Z1,Z2 : UCELL(Z1,Z2)=X].

IV THE LIST MODULE

Our goal is to design an abstract machine that provides its user with both unique and conventional list processing. This machine is the product of ULIST and a module LIST that we will specify next. The formal specification of LIST is given in Figure 3. The structure of this module is quite similar to ULIST: there is a single V-function CELL and four OV-functions CONS, CAR, CDR, and CONSP. However, there are important differences. First, whereas UCELL is a function from a UCAR-UCDR pair (X1,X2) to the unique list cell X--if any--with X1 and X2 as UCAR and UCDR, CELL is a predicate on the triple (X1,X2,X) that tests

```
module:   LIST:

  forall: Z1,Z2,Z

  vfns:   CELL(X1,X2,X)
           initial CELL(X1,X2,X)=false

  define: ISCELL(X) = (exists X1,X2 : CELL(X1,X2,X) and X~=?)

  ovfns:  CONS(X1,X2) -> X
           assert  ATOMP(X1) or ISCELL(X1)
                   ATOMP(X2) or ISCELL(X2)
           effects 'CELL(X1,X2,X)
                   not CELL(Z1,Z2,X)
                   X~=?
                   not ATOMP(X)
                   Z~=X => 'CELL(Z1,Z2,Z)=CELL(Z1,Z2,Z)
          CAR(X) -> X1
           assert  ISCELL(X)
           effects 'CELL(Z1,Z2,Z)=CELL(Z1,Z2,Z)
                   exists Z2 : CELL(X1,Z2,X)
          CDR(X) -> X2
           assert  ISCELL(X)
           effects 'CELL(Z1,Z2,Z)=CELL(Z1,Z2,Z)
                   exists Z1 : CELL(Z1,X2,X)
          CONSP(X) -> B
           effects 'CELL(Z1,Z2,Z)=CELL(Z1,Z2,Z)
                   B = ISCELL(X)
```

Figure 3. Specification of LIST Module

whether X is a conventional list cell with CAR X1 and CDR X2. This difference is necessary: because there may be more than one conventional list cell with a particular CAR and CDR, CELL cannot be a function. The second difference between the two modules is in the effects of UCONS and CONS. UCONS does not always change the ULIST state, but CONS always changes the LIST state. Even if there are already X1, X2, and X~=? such that CELL(X1,X2,X), an execution of CONS(X1,X2) will create a new cell with this CAR and CDR.

V PROPERTIES OF ULIST AND LIST

Even though we have not yet implemented ULIST x LIST, we can prove properties of this machine just on the basis of its specifications. We illustrate this point by proving three results: that UCELL is one-to-one, that two structurally isomorphic unique lists are identical, and that if two conventional lists are structurally isomorphic, then certain corresponding unique lists are identical.

Consider the claim that the specification of ULIST is consistent, that is, implementable. For example, if UCELL is not one-to-one on that part of its domain that does not map to "?", then the specifications for UCAR and UCDR are not realizable. For suppose X=UCELL(Z1,Z2)=UCELL(Z3,Z4)~=?. If Z1 is not equal to Z3, then UCAR(X) is required to return both Z1 and Z3, an impossibility. Similarly, if Z2 is different from Z4, then UCDR(X) is required to return two different values.

This result is not, by itself, sufficient to show that ULIST can be implemented. On the other hand, a provably correct implementation of ULIST--given below--implies this result. However, the result is easy to state and has interesting consequences. Moreover, its proof illustrates an general proof technique applicable to abstract machines.

Theorem 1. forall Z1,Z2,Z3,Z4 :
UCELL(Z1,Z2)=UCELL(Z3,Z4)~=?
=> Z1=Z3 and Z2=Z4

Proof. The theorem will be proved by induction on sequences of states of ULIST. This method of proving properties of abstract machines, which we call generator induction, is discussed in [9], [30], and [3]. We must prove the theorem for any initial state of ULIST and for any state S' such that the theorem holds in a state S and S' is a state resulting from executing a ULIST instruction in S.

The basis of the induction is immediate, since UCELL is always "?" in an initial machine state. Thus it suffices to assume the theorem

holds in some state S and to deduce its validity in a successor state S' that results from the operation UCONS(X1,X2) (since UCONS is the only operation that changes UCELL's results). Suppose that this execution of UCONS returns X and that, in the resulting state, there are Z1, Z2, Z3, and Z4 such that 'UCELL(Z1,Z2)='UCELL(Z3,Z4)~=?. If there has been no state change--the "else clause" of the effects--or if 'UCELL(Z1,Z2)=UCELL(Z1,Z2) and 'UCELL(Z3,Z4)=UCELL(Z3,Z4), then the inductive assumption gives the desired result. Suppose that there has been a state change--the "then clause" and that 'UCELL(Z1,Z2)~=UCELL(Z1,Z2). Thus Z1=X1 and Z2=X2. If Z3=X1 and Z4=X2, we are done. If Z3~=X1 or Z4~=X2, then the second equation of the else clause implies that 'UCELL(Z3,Z4)=UCELL(Z3,Z4) which is not equal to X by the first equation of the else clause, a contradiction. This completes the proof.

Next, we extend this result to show that structural equality implies identity for ULIST lists. Let us write UCAR*, UCDR*, and UCONSP* to refer to the values returned by these instructions in some state. (We introduce this notation to emphasize the careful distinction between these values, useful in stating static mathematical properties of a specification, and the instructions that might be executed to obtain them in some implementation.) Theorem 1 implies that, in any state, UCAR* and UCDR* are functions mapping the set of X such that UCONSP*(X) to a range not containing "?". It is a simple matter, using Theorem 1, to show that the conclusion X=Y follows from the hypotheses UCONSP*(X), UCONSP*(Y), UCAR*(X)=UCAR*(Y), and UCDR*(X)=UCDR*(Y); we leave the details to the reader. Using this corollary, we can prove that structural isomorphism implies identity for the lists of the ULIST machine. We define structural isomorphism of unique lists recursively by:

```
UISO(X,Y) <= if UCONSP*(X) and UCONSP*(Y)
                then UISO(UCAR*(X),UCAR*(Y))
                 and UISO(UCDR*(X),UCDR*(Y))
                else X=Y,
```

and can then prove:

Theorem 2. forall X,Y : UISO(X,Y) => X=Y.

Discussion. We wish to prove this theorem by structural induction on unique lists. (Structural induction is described by Burstall in [3]; the theorem can also be proved, less easily, by generator induction.) We will prove the theorem for the atoms that form the leaves of a unique list and we will prove that if the theorem holds for the proper sublists of a unique lists, then it holds for the entire list. For such an induction to be sound, it is essential that there be no circular lists. Fortunately, the ULIST machine instructions provide no way to create circular lists. Since there are no lists in an initial ULIST state, since each instruction creates at most one new list, and since we are only interested in machine states achievable by the execution of finitely many instructions, this induction is well-founded. (This same argument demonstrates that UISO is total.)

Proof. The basis of the induction is the case that ~UCONSP*(X) or ~UCONSP*(Y), and it is an immediate consequence of the definition of UISO. We make the inductive assumptions UCONSP*(X), UCONSP*(Y), UISO(UCAR*(X),UCAR*(Y)) => UCAR*(X)=UCAR*(Y), and UISO(UCDR*(X),UCDR*(Y)) => UCDR*(X)=UCDR*(Y), and must prove that UISO(X,Y) => X=Y. Expanding the definition of UISO(X,Y), we conclude that UISO(UCAR*(X),UCAR*(Y)) and UISO(UCDR*(X),UCDR*(Y)); hence by the inductive assumptions, UCAR*(X)=UCAR*(Y) and UCDR*(X)=UCDR*(Y). The corollary of Theorem 1 now yields the desired result.

Next we prove a theorem about a program that might be run by a top-level user of the ULIST x LIST machine to translate conventional lists to unique lists. (Because of the overhead associated with the maintenance of unique lists, it is common to do some computations with the corresponding conventional lists, and convert only the final result to a unique list.) We claim that this may be done by the program UCOPY defined as follows:

```
UCOPY(X) <= if CONSP(X)
            then UCONS(UCOPY(CAR(X)), UCOPY(CDR(X)))
            else X.
```

The major result about UCOPY is that if two conventional lists are isomorphic, then their U-copies are identical. Isomorphism of conventional lists is defined by:

```
ISO(X,Y) <= if CONSP*(X) and CONSP*(Y)
              then ISO(CAR*(X),CAR*(Y))
               and ISO(CDR*(X),CDR*(Y))
              else X=Y.
```

Let UCP(XA,XB) be an abbreviation for ISO(XA,XB) => UCOPY(XA)=UCOPY(XB). We will then prove:

Theorem 3. forall XA,XB : UCP(XA,XB)

Discussion. The meaning of this formula is subtle, since the effects and result of UCOPY are contingent upon the machine state in which it is executed. A more precise statement would be as follows. Suppose ISO(XA,XB). Suppose that UCOPY is applied to XA, beginning in some state S1. This application terminates (since our lists are acyclic) yielding a value XA' and a state S2. Suppose, moreover, that S3 is any successor of S2, resulting from a series of state transitions, starting at S2. Finally, suppose that UCOPY, applied to XB from state S3, yields value XB' and state S4. Then XA'=XB'.

We are going to prove this result by induction. An inductive assumption of this rather lengthy form would be very cumbersome. Fortunately, the machine specifications imply that if UCONSP*(X) holds in some state S, then it holds in every successor state. Also, if UCAR*(X) and UCDR*(X) are defined in a state S, then they remain defined and retain the same value in all successor state. In view of these facts, sometimes called frame axioms, we can safely omit further references to changing states and use the simple statement of the theorem. (It is very interesting to consider whether this kind of problem reduction might be done mechanically.)

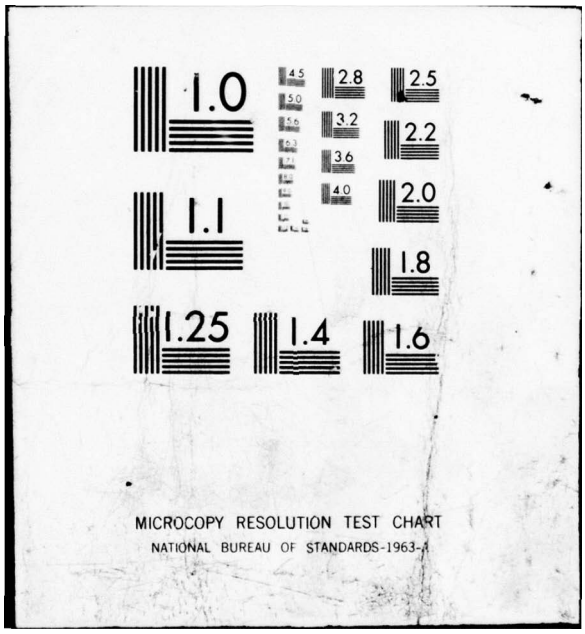
Proof. First, suppose ~CONSP*(XA). Then ISO(XA,XB) implies that XA=XB. Also, UCOPY(XA)=XA and UCOPY(XB)=XB so that the desired result is immediate. Next suppose CONSP*(XA). Then ISO(XA,XB) implies that CONSP*(XB). We proceed by simultaneous structural induction on XA and XB. That is, we assume

(I1) forall X : UCP(s(XA),X)

AD-A050 154 SRI INTERNATIONAL MENLO PARK CALIF F/G 9/2
THE SEMIAUTOMATIC GENERATION OF INDUCTIVE ASSERTIONS FOR PROVIN--ETC(U)
NOV 77 B ELSPAS, R S BOYER, K N LEVITT F44620-73-C-0068
UNCLASSIFIED AFOSR-TR-78-0114 NL

2 OF 2
AD
A050 154

END
DATE
FILMED
3-78
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(I2) forall X : UCP(X,s(XB))

where s is either CAR* or CDR*. (Clearly, UCP is symmetric in its two arguments.) From ISO(XA,XB) it follows that

(I3) ISO(CAR*(XA),CAR*(XB)), and

(I4) ISO(CDR*(XA),CDR*(XB)).

Combining these results with I1 and I2 we obtain

(I5) UCOPY(CAR*(XA))=UCOPY(CAR*(XB)), and

(I6) UCOPY(CDR*(XA))=UCOPY(CDR*(XB)).

We must prove that UCOPY(XA)=UCOPY(XB). This is done as follows:

```
UCOPY(XA)=UCONS*(UCOPY(CAR*(XA)), UCOPY(CDR*(XA)))
          {by the definition of UCOPY}
         =UCONS*(UCOPY(CAR*(XB)), UCOPY(CDR*(XB)))
          {by I5, I6}
         =UCOPY(XB)
          {by the definition of UCOPY}.
```

VI IMPLEMENTATION OF ULIST x LIST

We now wish to implement the machine specified above in terms of more primitive facilities. Specifically, we will consider a machine, LIST x SEARCH, that has conventional list processing capabilities and an associative search capability. Since we retain the LIST facilities in this second level machine, the main problem is to describe ULIST in terms of associative search and conventional list processing.

Our SEARCH machine is formally specified in Figure 4. The basic idea is that there are a number of "search tables"--a special table called PRIMARYTABLE that exists initially, and as many secondary tables as the user wishes to create using the NEWTABLE instruction. In each table one can save a value under a key, writing over any previously saved value, or look up the value saved under a key. (We could simplify the argument structure and specifications of the instructions of this module by using only a single table and a GETOP instruction whose single argument combined the information in the two arguments of the present

```
module:   SEARCH:
  forall: K,T
  vfns:   PRIMARYTABLE()
           initial PRIMARYTABLE()~=?
          GET(KEY,TABLE)
           initial GET(KEY,TABLE)=?
          TABLEP(TABLE)
           initial TABLEP(TABLE) = (TABLE=PRIMARYTABLE())

  ovfns:  NEWTABLE() -> TABLE
           effects 'PRIMARYTABLE()=PRIMARYTABLE()
                   'GET(K,T) = GET(K,T)
                   'TABLEP(T) = (TABLEP(T) or T=TABLE)
                   not TABLEP(TABLE)
                   TABLE~=?
          SAVE(VALUE,KEY,TABLE)
           assert  TABLEP(TABLE) and VALUE~=? and KEY~=?
           effects 'PRIMARYTABLE()=PRIMARYTABLE()
                   'GET(K,T) = if K=KEY and T=TABLE
                                  then VALUE
                                  else GET(K,T)
                   'TABLEP(T)=TABLEP(T)
          GETOP(KEY,TABLE) -> VALUE
           assert  TABLEP(TABLE) and KEY~=?
           except  NOTTHERE: GET(KEY,TABLE)=?
           effects 'PRIMARYTABLE()=PRIMARYTABLE()
                   'GET(K,T)=GET(K,T)
                   'TABLEP(T)=TABLEP(T)
                   VALUE=GET(KEY,TABLE)
          PRIMARYTABLEOP() -> TABLE
           effects TABLE='PRIMARYTABLE()=PRIMARYTABLE()
                   'GET(K,T)=GET(K,T)
                   'TABLEP(T)=TABLEP(T)
```

Figure 4. Specification of SEARCH

GETOP. But we believe that the version given yields a clearer implementation and it is also more suggestive of the implementation used by Deutsch [6].)

Note that the specification of GETOP uses an exception if there is no entry in the table under the key that is sought. It is worthwhile to contrast the use of exceptions and assertions in this specification. We assert that TABLEP(TABLE), indicating that a compilation or verification

assert that TABLEP(TABLE), indicating that a compilation or verification may assume this fact in processing the implementation of GETOP but must verify it for uses of GETOP. The assertion describes a condition that must be guaranteed to hold at calls of the function. The exception, on the other hand, describes a condition--the presence of a particular entry in the table--that may or may not hold. Implementation programs are simplified by the possibility of structuring them to consider the "normal" and "exceptional" cases separately.

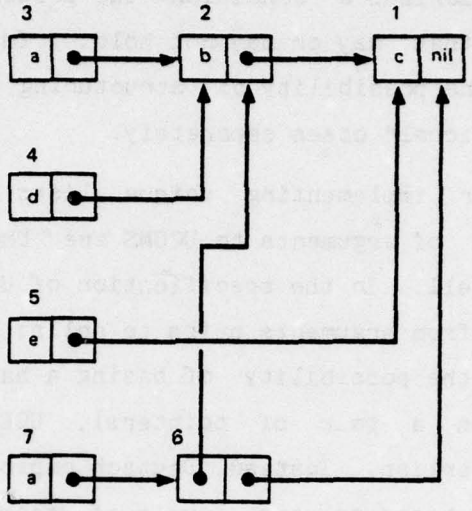
Any scheme for implementing unique lists requires determining whether a given pair of arguments to UCONS are the UCAR and UCDR of a previously UCONSed cell. In the specification of ULIST, the V-function UCELL serves to map from arguments pairs to cells; however, because of a quirk of Interlisp (the possibility of basing a hash probe on a single pointer but not on a pair of pointers), UCELL is not directly implementable in Interlisp. Instead, Deutsch employed--and we formally describe--a scheme based on two levels of search. This scheme is illustrated in Figure 5c. The lists in Figure 5a are shown as they might be uniquely represented in Figure 5b. Each list cell is shown with a numeric cell identifier corresponding to the list number. The primary table of Figure 5c represents the association between UCDRs of cells and secondary tables. A secondary table, corresponding to a particular UCDR X2, then associates UCARs of cells with the cell, if any, having that UCAR and UCDR X2.

Implementation of ULIST x LIST by LIST x SEARCH is now possible. The implementation has three parts: a representation of states, an initialization program, and programs realizing each of the OV-functions of the upper level. First, the state representation is described by two formulas:

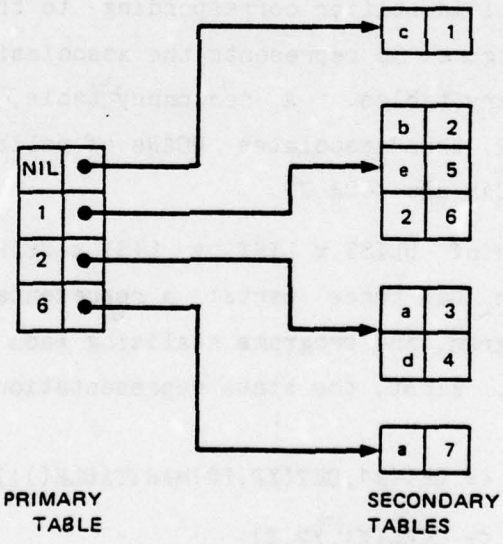
```
UCELL(X1,X2)  <= GET(X1,GET(X2,PRIMARYTABLE()))

CELL(X1,X2,X) <= CELL(X1,X2,X)
                 and GET(X1,GET(X2,PRIMARYTABLE()))~=X
```

1. (c)
2. (b c)
3. (a b c)
4. (d b c)
5. (e c)
6. ((b c) c)
7. (a (b c) c)

(a) A SET OF SEVEN LIST STRUCTURES



(b) THE SAME LIST STRUCTURES IMPLEMENTED WITH SHARING OF COMMON CELLS



(c) THE SEARCH TABLES FOR THESE LISTS

Figure 5. Implementation of ULIST

In these formulas, the left-hand sides refer to V-functions of the upper machine and the right-hand sides to V-functions of the lower machine. The first of these formulas describes the state correspondence needed to implement the two level search procedure: it provides that an upper state where UCELL(X1,X2) is not "?" will be represented by a lower state where the double search with X1 and X2 yields a result other than "?". Also, it requires that if UCELL(X1,X2) is "?", then the double search in the lower level state must yield "?" too.

The second formula describes how the upper level conventional lists are represented. The answer is, they are represented by lower level lists. We state this by using CELL on both sides of the definition, to be read as describing the upper level CELL in terms of the lower level CELL. However, there is a subtlety: some of the lower level lists will be used in the implementation to represent upper level unique lists and these, so far as the upper level is concerned, do not exist as lists. Thus, we add the second conjunct to this formula to exclude these lists from the set of upper level cells.

The initialization program must start from an initial state of LIST x SEARCH and arrive at a state of that machine that represents an initial state of ULIST x LIST. However, all that is required of an initial state of ULIST x LIST is that UCELL(X1,X2) is "?" and CELL(X1,X2,X) is "false" and, in view of the representation just given, this is represented by an initial state of LIST x SEARCH. Hence, the empty program suffices for initialization.

Finally, we must realize the upper level OV-functions UCONS, UCAR, UCDR, UCONSP, CONS, CAR, CDR, and CONSP by lower level programs. These are given in Figure 6. First, note that occurrences of CONS, CAR, CDR, and CONSP in the defining programs denote these instructions on the lower level machine. Next, note that the defining programs for the top level functions CONS, CAR, and CDR are trivial because exactly the right instruction exists at the lower level. The defining programs for UCAR and UCDR are also single instructions; this is a consequence of the

```
UCONS(X1,X2)  : begin locals TABLE, C;
                execute TABLE <- GETOP(X2,PRIMARYTABLEOP()) then
                  on normal:   execute C <- GETOP(X1,TABLE) then
                                 on normal:   return(C);
                                 on NOTTHERE: C <- CONS(X1,X2);
                                              SAVE(C,X1,TABLE);
                                              return(C) end;
                  on NOTTHERE: TABLE <- NEWTABLE();
                               C  <- CONS(X1,X2);
                               SAVE(C,X1,TABLE);
                               SAVE(TABLE,X2,PRIMARYTABLEOP()) end end;
UCAR(X)       : CAR(X);
UCDR(X)       : CDR(X);
UCONSP(X)     : begin locals TABLE, C;
                  if CONSP(X)
                     then execute
                            TABLE <- GETOP(CDR(X),PRIMARYTABLEOP()) then
                            on normal:
                                 execute C <- GETOP(CAR(X),TABLE);
                                 then
                                   on normal:   return(C=X);
                                   on NOTTHERE: return(false) end;
                            on NOTTHERE: return(false) end
                    else return(false) end;
CONS(X1,X2)   : CONS(X1,X2);
CAR(X)        : CAR(X);
CDR(X)        : CDR(X);
CONSP(X)      : CONSP(X) and ~UCONSP(X);
```

Figure 6. Top-Level Machine Implementation

decision we made to represent upper level unique lists by lower level conventional lists. The non-trivial implementations are those for UCONS, UCONSP, and CONSP.

The implementation for UCONSP is a block that introduces two local variables: TABLE and C. If the argument X does not satisfy the lower level CONSP predicate, it cannot--in view of the representation--satisfy UCONSP. However, if it is a list cell, the UCONSP program uses the "execute" statement, a special feature of our implementation language. We use this statement to call an OV-function that may have exceptions and then deal with the normal exit and the exceptional exits in turn.

Thus the UCONSP program first searches in the primary table with CDR(X) as key. If there is no exception, then the TABLE that results is searched with CAR(X) as key. A normal exit from this second search with result C indicates that C is a unique list with the same components as X and therefore is the only such unique list. Hence X is a unique list if and only if it is C. If either search has an exceptional exit, this means that there is no unique list with CAR(X) and CDR(X) as components. Thus UCONSP returns "false".

The implementation of UCONS has a similar structure. If both searches have normal exits and result C, then UCONS just returns C. If the first search encounters the NOTTHERE exception, this means that there are no existing unique lists with UCDR X2. Hence we create a new search table to record such unique lists, enter it in the primary table under key X2, create the new (representation of a) unique list CONS(X1,X2), and enter it in the new secondary table under key X1. The new list is then the answer returned by UCONS.

If the first search has a normal exit, but the second search has a NOTTHERE exception, this indicates that there is already a secondary search table TABLE for unique lists with UCDR X2, but that there is no entry in TABLE with X1 as UCAR. Hence we again create a new unique list representation CONS(X1,X2), enter it in TABLE under key X1, and return it as the answer of UCONS.

Finally, the implementation of CONSP introduces some difficulty. Although there is a CONSP instruction at the lower level, it does not suffice: the lower level CONSP is satisfied by the lower level cells that represent upper level unique lists but these are not conventional lists in the abstraction provided by the upper level machine. We have given an implementation that makes an additional test [~UCONSP(X)] to avoid this problem, a correct but unpleasantly inefficient implementation of what ought to be a low-overhead type checking operation. For present purposes, the correct but inefficient implementation suffices; Section IX discusses some alternatives.

VII CORRECTNESS OF THE IMPLEMENTATION

The proof that an implementation is correct with respect to a pair of machine specifications and a state representation has two parts. First, we must prove that the initialization program for the lower level--in this case the empty program--can be executed from any initial state of the lower level machine to yield a lower level state that represents an initial state of the upper level machine. Second, we must prove that this representation is preserved by the execution of the implementations of OV-functions in the lower machine. That is, suppose S and S' are states of the upper machine and T and T' are states of the lower machine. Suppose that S' is a state that results from the execution of an OV-function call "F(X)" according to the specification of the upper machine. Also, suppose that T' is a state that results from the execution of the implementation of "F(X)" in the lower machine. Then, we must prove that T' is a representation of S'. (This may be thought as a proof that the diagram of Figure 6 commutes.)

In doing these proofs, it is important to note that the execution of the implementations of the upper machine instructions does not fully exercise the facilities of the lower machine. For example, in our LIST x SEARCH machine there are states such that, for some X, the result of "GETOP(X,PRIMARYTABLEOP())" is neither an exception nor a secondary table but, instead, a list cell. Since we never store anything other than secondary tables in the primary table, we know that this can never occur and would like to use this knowledge to help the proof. We can do this by formulating an invariant predicate I(T) on the states T of the lower machine. We then prove that I holds for states resulting from the initialization of the lower machine. We also prove that if I(T) holds, and P is the implementation of an upper machine OV-function, then I(T') holds where T' results from T when P is executed in the lower machine. Having proved such an invariant property, we may assume that I holds for all states that arise in the proofs described in the previous paragraph.

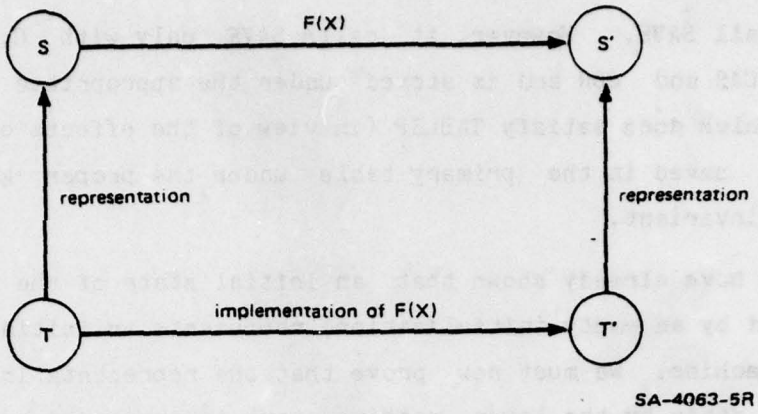


Figure 7. A Necessary Condition for Implementation Correctness

We will illustrate the proof of correctness of implementations in this methodology by proving the correctness of the implementation given in the preceding section. The necessary invariant assertion has two parts. First, if a fetch from the primary table yields a result, not "?", then that result is a (secondary) table. Second, if a fetch from the secondary table yields a result, not "?", then that result is a list cell whose components are the keys of the two fetches. Stating this formally, we have

```
I(T) = GET(Z2,PRIMARYTABLE())=TABLE~=?
       implies (TABLEP(TABLE) and
                (GET(Z1,TABLE)=Z~=?
                 implies (CONSP(Z) and CAR(Z)=Z1 and CDR(Z)=Z2))).
```

(Note that the notation is such that the state T does not appear explicitly in the right-hand side of this definition; note also the implicit universal quantification of Z1, Z2, Z, and TABLE.) It is easy to show that I is an invariant of the lower machine states that arise in the implementation. It is true of the initial state because its antecedent is always false in this initial state. If it is true of a state T, then the execution of the implementations of UCAR, UCDR, UCONSP, CONS, CAR, CDR, and CONSP involve no calls of SAVE and therefore no changes in GET. The remaining case is the implementation of UCONS(X1,X2). This implementation can affect the truth of I because it

does call SAVE. However, it calls SAVE only with C, which has the proper CAR and CDR and is stored under the appropriate keys, and with TABLE which does satisfy TABLEP (in view of the effects of NEWTABLE) and is also saved in the primary table under the proper key. Thus I is indeed invariant.

We have already shown that an initial state of the lower machine, followed by an empty initialization, represents an initial state of the upper machine. We must now prove that the representation of the upper machine state by the lower machine state is preserved by the execution of the implementations. It should be clear, in each case, that the result returned satisfies the corresponding specification. Except for UCONS and CONS, the instructions of the upper machine are implemented by programs whose only effect is the return of a result; thus these implementation all preserve the representation of the specified upper state by the resulting lower state.

The upper machine's CONS instruction is implemented by the CONS of the lower machine. Since the execution of the lower machine CONS affects only the V-function CELL, and only in a way consistent with the representation, this implementation also preserves the representation. The remaining upper machine construction is UCONS; consider its implementation. If both execute statements are "normal", there is no state change; that the result returned is correct is immediate from the invariant. If the outer execute statement has a normal exit and the inner a "NOTTHERE" exception, then the implementation creates exactly one new cell, and saves it in the proper table, thus preserving the representation of UCELL. Moreover, since the first conjunct of the representation of CELL becomes true exactly where the second conjunct becomes false, the specification that the representation of the upper level CELL be unaffected by execution of UCONS is satisfied.

Finally, if both execute statements have "NOTTHERE" exceptions, then a new secondary table is created and saved under the proper key in the primary table. This does not affect the representation, and the

remainder of code in this case preserves the representation by the argument just made for the case of a single exception.

This completes the proof of the HASH-CONS implementation.

VIII FURTHER IMPLEMENTATIONS

The preceding sections have described how properties of an unimplemented machine can be proved from its formal specifications, how such a machine can be realized in terms of a more primitive machine, and how such a realization can be proved with respect to the two machines. To save space, we will in this section sketch rather than fully presenting the further refinement of the LIST x SEARCH machine.

If Interlisp is an acceptable primitive machine, then the programs described so far solve the original problem, since it provides the LIST and SEARCH facilities to which we have reduced the problem. This would raise an interesting problem for the proof of the LIST x SEARCH specifications. The most complete extant specification of Interlisp, [19], is not written in SPECIAL; this proof would thus require a different theory from that discussed here.

A more primitive Lisp than Interlisp can also be used as the basis of our hierarchy. For example, one can easily implement the facilities of SEARCH, except for the PRIMARYTABLEOP instruction, in terms of Lisp lists; the implementation is just the usual Lisp "association list". The implementation of PRIMARYTABLEOP can be accomplished by using a single variable to remember which association list represents the primary search table. That is, LIST x SEARCH can be implemented in terms of VARIABLE x LIST. (VARIABLE is a very simple module: its state is the value saved in the variable and it has two instructions, one to read the value and one to save a new value.) The machines of this hierarchy, and their component modules, are shown in Figure 8.

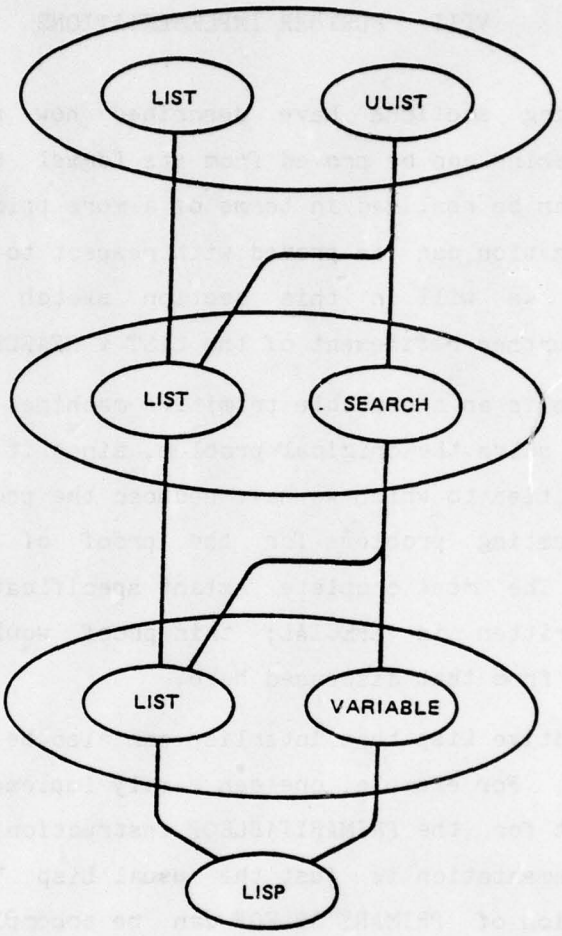


Figure 8. An Implementing Hierarchy for Unique Lists

Alternatively, one can distinguish two kinds of search operations in the implementation of ULIST--those that start from the primary table and those that start from one of the secondary tables. Actual use of ULIST suggests that it is reasonable to use a hashtable for the primary search table and alists for the secondary tables. Since Interlisp provides named hashtables, this means that LIST x SEARCH could be realized by HASHTABLE x LIST and, in turn, HASHTABLE x LIST could be realized by Interlisp. (We will not provide an implementation of HASHTABLE here. The interested reader should consult [30]; in that paper, HASHTABLE is implemented in terms of arrays and a "hash probe" function and it is proved that the implementation is correct .)

IX CONCLUDING REMARKS

In Section VI, we implemented some specifications in an Algol-like language called ILPL, which is described in Appendix C of [22]. However, the use of this language, while convenient, is not essential to the use of our methodology. On the contrary, we believe that enough structure is given to even a large system by its decomposition and precise specification in SPECIAL to permit implementation in many languages. The critical points in the design and implementation of systems tend to be global issues such as a decisions on how to decompose a system into modules or how to describe the implementation of a module by a hierarchy of abstractions--exactly the areas in which SPECIAL is expressive. By contrast, the details of any particular programming language usually address very local issues in programming, e.g. whether to use a case or conditional statement to describe a choice, whether to use a "while-do" or a "repeat-until" statement to describe a particular iteration. While such local decisions certainly have an impact on the clarity of the programs that can be written, we believe that this impact is negligible by comparison with the impact of well or poorly done overall design and specification. If the latter is precise, so that a

large system is implementable by a large number of loosely coupled small parts, then many different languages may be equally good for implementing the parts.

This is not to deny that care should be taken in the choice of an implementation language. Certainly one ought to use a language with lucid syntax and a flexible set of control structures. Since we advocate the decomposition of a program into many parts, it follows that we recommend choosing a language that can be compiled into a form in which linkage between the parts is economical. Since we seek to implement systems, we are interested in the ultimate efficiency of implementations and therefore require a language in which machine-level representations can be described for the use of the most primitive levels of a hierarchy.

A related issue is the provision of data structures by the base language. For example, we assumed above that our base machine provided a set of objects satisfying the predicate ATOMP and disjoint sets of objects to represent abstractions such as cells and tables. If an adequate facility for defining concrete data types is present in the base, then it need not be provided by the hierarchy and--if the base language is carefully implemented--the cost of soundly manipulating objects of different types will be kept to a minimum. (Such a base should permit an efficient implementation of the upper level CONSP instruction in Section VI; by contrast, [6] is not intended as a hierarchical solution to the unique list problem, does not distinguish the list cells of the different levels of abstraction, and uses the same selectors CAR and CDR for all of them.) If the base doesn't have a sufficient facility, for example because it is a bare machine, then a type system must be synthesized as part of the hierarchy of machines. This can be quite hard, but it is possible [22].

Some base languages will provide not only concrete but also abstract data structures; these include CLU [15], a modification of Simula [23], Modula [31], and Alphard [32]. Some of these facilities

are clearly redundant if our methodology is used with a tool that statically confirms that implementation programs are compatible with specifications, e.g. in what functions they call or what objects they refer to. On the other hand, the use of such a base language can ease the proof that implementation programs have the protection semantics implicit in the methodology.

Boyer and Moore have developed a formal semantics and a verification condition generator for our methodology [1], using the underlying theory of their Lisp Theorem Prover [2]. This makes it possible to produce precise machineable versions of the theorems given in Section VII and preliminary experiments encourage us in the hope that these theorems may be mechanically proved. This will be a major theme in our future work.

There are certainly many other ways to specify programs formally. We think that the method of algebraic specifications [8] is very promising. It is similar to our method in its precision and compatibility with formal proof. It appears, in some published examples, to produce specifications that are quite concise but may require of readers greater mathematical sophistication than do ours; we are not aware of its use on examples as large as [22]. It would be premature to draw firm conclusions about the relative merits of the two methods and we look forward to the further development of both.

REFERENCES

1. Boyer, R. S. and Moore, J S. Private Communication (June 1977).

2. Boyer, R. S. and Moore, J S. A Lemma Driven Automatic Theorem Prover for Recursive Function Theory, Proceedings of the International Joint Conference on Artificial Intelligence, (Cambridge, Mass., August 1977).

3. Burstall, R. M. Proving Properties of Programs by Structural Induction, Computer Journal, 12, 1 (Jan. 1969), pp. 41-48.

4. Dahl, O. J., Myhrhaug, B., and Nygaard, K. Common Base Language, S-22, Norwegian Computing Center, Oslo, Norway (Oct. 1970).

5. Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R. Structured Programming, (Academic Press, New York, N.Y., 1972).

6. Deutsch, L. P. An Interactive Program Verifier, Ph.D. Thesis, Dept. of Computer Science (University of California at Berkeley, 1973).

7. Good, D. I. Provable Programming, Proceedings of International Conference on Reliable Software, SIGPLAN Notices, 10, 6 (June 1975), pp. 411-419.

8. Guttag, J. Abstract Data Types and the Development of Data Structures, C. ACM, 20, 6 (June 1977), pp. 396-404.

9. Hoare, C. A. R. Proof of Correctness of Data Representations, Acta Informatica, 1, 4 (1972), pp. 271-281.

10. Hoare, C. A. R. and Wirth, N. An Axiomatic Definition of the Programming Language PASCAL, Acta Informatica, 2, 4 (1973), pp. 335-355.

11. Ichbiah, J. D. et al. The System Implementation Language LIS, Technical Report 4549 E/EN (December 1974), Compagnie Internationale pour l'Informatique, Louveciennes, France.

12. Igarashi, S., London, R. L., and Luckham, D. C. Automatic Program Verification I: A Logical Basis and its Implementation, Acta Informatica, 1, 4 (1975), pp. 145-182.

13. An Appraisal of Program Specifications, Computation Structures Group Memo 141-1 (April 1977), Laboratory for Computer Science, M. I. T., Cambridge, Massachusetts.

14. Liskov, B. and Zilles, S. Programming with Abstract Data Types, Proceedings of ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN Notices, 9, 4 (Apr. 1974), pp. 50-59.

15. Liskov, B. and Zilles, S. Specification Techniques for Data Abstraction, IEEE Trans. Software Engineering, SE-1, 1 (March 1975), pp. 7-19.

16. McCarthy, J. A Basis for a Mathematical Theory of Computation. In Computer Programming and Formal Systems, Braffort and Hirschberg (Eds.), pp. 33-70 (North-Holland, Amsterdam, 1963).

17. McCarthy, J. et al. LISP 1.5 Programmer's Manual, (The M. I. .T. Press, Cambridge, Mass., 1962).

18. Manna, Z., Ness, S., Vuillemin, J. Inductive Methods for Proving Properties of Programs, C. ACM, 16, 8 (Aug. 1973), pp. 491-502.

19. Moore, J S. The Interlisp Virtual Machine Specification, Xerox Palo Alto Research Center, Report CSL 76-5 (Sept. 1976).

20. Morris, J. Protection in Programming Languages, C. ACM, 16, 1 (Jan. 1973), pp. 15-21.

21. Morris, J. M. Types Are Not Sets, ACM Symposium on Principles of Programming Languages, Boston, Mass. (Oct. 1973), pp. 120-124.

22. Neumann, P. G. A Provably Secure Operating System: The System, Its Applications, and Proofs, Final Report, SRI Project 4332 (February 1977), SRI International, Menlo Park, California.

23. Palme, J. Protected Program Modules in Simula 67, Research Institute of National Defense, Stockholm, Sweden (July 1973).

24. Parnas, D. L. A Technique for Software Module Specification with Examples, C. ACM, 15, 5 (May 1972), pp. 330-336.

25. Parnas, D. L. On the Criteria to be Used in Decomposing Systems into Modules, C. ACM, 15, 12 (Dec. 1972), pp. 1053-1058.

26. Robinson, L. and Levitt, K. N. Proof Techniques for Hierarchically Structured Programs, C. ACM, 20, 4 (Apr. 1977), pp. 271-283.

27. Robinson, L. et al. On Attaining Reliable Software for a Secure Operating System, Proceedings of International Conference on Reliable Software, SIGPLAN Notices, 10, 6 (June 1975), pp. 267-284.

28. Roubine, O. and Robinson, L. SPECIAL Reference Manual (3rd edition), Technical Report CSL-45, SRI Project 4828 (January 1977), SRI International, Menlo Park, California.

29. Wegbreit, B. The Treatment of Data Types in EL1, C. ACM, 17, 5 (May 1974), pp. 251-164.

30. Wegbreit, B. and Spitzen, J. M. Proving Properties of Complex Data Structures, J. ACM, 23, 2 (Apr. 1976), pp. 389-396.

31. Wirth, N. Modula: A Language for Modular Multiprogramming, Software--Practice and Experience, 7 (1977), pp. 3-35.

32. Wulf, W. A. ALPHARD: Toward a Language to Support Structured Programs, Computer Science Department, Carnegie-Mellon University, (Apr. 1974).

33. Yourdon, E. and Constantine, L. L. Structured Design, (Yourdon Press, New York, New York, 1975).

Appendix B

PRIMITIVE RECURSIVE PROGRAM TRANSFORMATION

Robert S. Boyer
J Strother Moore
Robert E. Shostak

PRIMITIVE RECURSIVE PROGRAM TRANSFORMATION

R. S. Boyer,[1] J S. Moore,[2] and R. E. Shostak[3]

ABSTRACT

We describe how to transform certain flowchart programs into equivalent explicit primitive recursive programs. The input/output correctness conditions for the transformed programs are more amenable to proof than the verification conditions for the corresponding flowchart programs. In particular, the transformed correctness conditions can often be verified automatically by the theorem prover developed by Boyer and Moore [1].

KEY WORDS

flowcharts, LISP, program verification, structural induction, theorem proving.

INTRODUCTION

Experiments with the theorem prover developed by R. Boyer and J Moore [1] have shown that structural induction in combination with symbolic evaluation and some generalization heuristics can be used to prove properties of a wide variety of LISP functions completely automatically. The key property of these functions making them amenable to induction is their explicit primitive recursive specification. Roughly speaking, the explicit primitive recursive form produces the effect that when the formula to be proved in the induction conclusion is symbolically evaluated, it assumes the form of the induction hypothesis.

In order to use the theorem prover on flowchart programs, it is necessary to translate the flowcharts into functional form. The easiest approach is that described in McCarthy [3] which produces partial recursive specifications. One is then forced either to extend the theorem prover to cope with a limited class of partial recursive specifications (as Moore does in [4]) or to further transform these specifications (where possible) into explicit primitive recursion. In this paper we are concerned with the second approach.

1 R. S. Boyer is employed by Stanford Research Institute, Menlo Park, California. This work was supported in part by ONR Contract No. N00014-75-C-0816.

2 J S. Moore is employed by Xerox Palo Alto Research Center, Palo Alto, California.

3 R. E. Shostak is employed by Stanford Research Institute, Menlo Park, California. This work was supported in part by AFOSR Contract No. F44620-73-C-0068.

Of course, not all programs compute primitive recursive functions (for example, programs that compute Ackermann's function or that interpret FORTRAN programs compute partial recursive, but not primitive recursive functions.) Furthermore, it is undecidable whether a function for which a partial recursive definition is given is primitive recursive. Thus, the method described here is not applicable to arbitrary flowchart programs, but only to those fitting certain schemes known to describe primitive recursive functions.

AN EXAMPLE

Our approach is best outlined with an example. Although we have restricted our presentation to the domain of lists and numbers, the general ideas are more broadly applicable.

Figure 1 shows a flowchart program computing the function `int(x)` that converts a binary number represented as a list of 1's and 0's into an integer. The program scans the input list from left to right. At each position scanned, it doubles the value of an accumulator A and adds the value of the scanned bit. After all bits have been scanned, the value of the accumulator is returned.

Consider the theorem stating that left-shifting a binary number (i.e., tacking a 0 onto the right end) has the effect of doubling that number's value:

(1) `int(append(L,list(0))) = 2*int(L),`

where L is understood to be a universally quantified variable ranging over all lists of 1's and 0's.

The first step in proving the theorem is to convert the flowchart program into functional form. McCarthy [3] has shown that one can do this in a mechanical way for arbitrary flowchart programs by introducing a new recursive function for each tag point. In the above example, one obtains:

```
int(x) = int1(x,0),
```

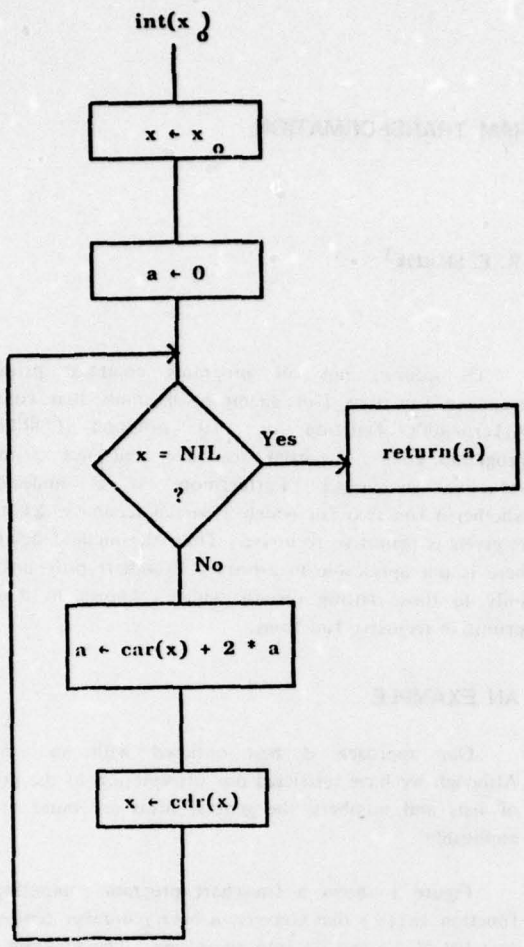


FIGURE 1

where

```
int1(x,a) =
          if x = NIL
             then a;
             else int1(cdr(x),car(x)+2*a).
```

The theorem to be proved can now be stated:

(2) `int1(append(L,list(0)),0) = 2*int1(L,0).`

One might now be tempted to try to prove (2) using structural induction on L. The basis case, L = NIL, goes through easily because both sides of (2) symbolically evaluate to 0. The induction step, however, does not go through. For that step, one assumes the induction hypothesis (1), and tries to prove:

(3) `int1(append(cons(B,L),list(0)),0)`
`=`
`2*int1(cons(B,L),0),`

where B is a variable ranging over the set {0, 1}.

Symbolically evaluating both sides of (3) gives:

(4) `int1(append(L,list(0)),B) = 2*int1(L,B).`

At this point, if all had gone well, we would have been able to invoke the induction hypothesis (2) and been done. But although (4) is similar to (2), it is not quite the same. Specifically, the second argument place of int1 is filled with 0 in the one case and with B in the other.

The source of the difficulty is that the form of the definition of int1 is not primitive recursive. In particular, the primitive recursive form requires all parameters but the "control" parameter (i.e., the one in the first argument position of int1) to be unmodified in the internal recursive calls. In the definition of int1, however, the second argument is changed from a to car(x)+2*a.

TRANSFORMATION TO PRIMITIVE RECURSION

The solution we propose here is to transform the non-primitive recursive definition of int1 into one that is primitive recursive. The transformation works on all functions that are instances of the scheme:

```
f(x,y) = if p(x) then g(x,y)
                 else f(n(x),h(x,y)),
```

where p, g, n, and h are primitive recursive.

The primitive recursive transform of f is f':

```
f'(x,y) =
      g(finalx(x),finaly(rev(seqx(x)),y))
```

where finalx, finaly, and seqx are primitive recursive functions whose definitions are exhibited below:

```
finalx(x) = if p(x) then x
                    else finalx(n(x)),

finaly(xl,y) =
             if xl=NIL
                then y
                else h(car(xl),
                       finaly(cdr(xl),y),

seqx(x) = if p(x) then NIL
                  else cons(x,seqx(n(x))),
```

and rev is the primitive recursive function which reverses a list:

```
rev(x) =
        if x=NIL then NIL
                 else append(rev(cdr(x)),
                             list(car(x))).
```

The justification for the theorem:

$$f(X_0,Y_0) = f'(X_0,Y_0)$$

is as follows: Let X_i denote $n^i(X_0)$ and let k be the smallest non-negative integer such that $p(X_k)$, then

(5) $$f(X_0,Y_0) = g(X_k, h(X_{k-1}, h(X_{k-2},\ldots,h(X_0,Y_0)\ldots))).$$

However,

$$finalx(X_0) = X_k$$

and

$$seqx(X_0) = (X_0\ X_1\ \ldots\ X_{k-1}).$$

Thus,

$$rev(seqx(X_0)) = (X_{k-1}\ X_{k-2}\ \ldots\ X_1\ X_0)$$

so that

$$finaly(rev(seqx(X_0)),Y_0) = h(X_{k-1},h(X_{k-2},\ldots,h(X_0,Y_0)\ldots)).$$

Therefore,

$$f'(X_0,Y_0) = g(X_k,h(X_{k-1},h(X_{k-2},\ldots h(X_0,Y_0)\ldots))),$$

which is just $f(X_0,Y_0)$ by (5).

Informally, seqx constructs a list of the successive values x will take on during the computation of f. This list, in reverse order, is then given to finaly which computes the final value of the "accumulator" y. This value, and that of finalx which is the final value of x, is then given to g to compute the final output of f.

In the special case where p(x) is x=NIL, n(x) is cdr(x), and h(x,y) can be expressed as a function, h', of car(x) and y the transform is simpler:

$$f'(x,y) = g(NIL,finaly(rev(x),y)),$$

where we use h' for h in the definition of finaly. The informal justification of this is that if the final y can be computed only in terms of the car's of the successive values of x, then we need not compute the sequence of x values but merely the sequence of car(x) values. But if p and n are as above this sequence is just x.

It is easy to see that int1 is an instance of the scheme described by f, and in fact is an example of the simpler case, since we can let:

```
p(x) = x=NIL
g(x,y) = y
n(x) = cdr(x)
h(x,y) = car(x) + 2*y.
```

Thus, we get:

```
int1'(x,a) = finala(rev(x),a).
```

where finala is:

```
finala(x,a) =
            if x=NIL
               then a
               else
               car(x)+2*finala(cdr(x),a).
```

Given this definition of int1', the example theorem:

(2) int1(append(L,list(0)),0) = 2*int1(L,0),

becomes:

```
finala(rev(app(L,list(0))),0)
                =
        2*finala(rev(L),0).
```

While this theorem is somewhat more complicated (syntactically) than (2), all of the functions in it are primitive recursive and it can be proved immediately by the theorem prover described in [1].

DISCUSSION

The idea that flowchart programs can sometimes be replaced by equivalent explicit primitive recursive functions was first mentioned in the 1934 work of R. Peter [5]. To quote from Peter ([5], pp. 69):

"5. It may be seen in a similar way that in general a recursion of the form

$$\varphi(0,a) = \alpha(a),$$

$$\varphi(n+1,a) = \beta(n,a,\varphi(n,\gamma_1(n,a)),\varphi(n,\gamma_2(n,a)),\ldots,\varphi(n,\gamma_k(n,a)))$$

and even a definition of the form

$$\varphi(0,\alpha_1,\ldots,\alpha_r) = \alpha(\alpha_1,\ldots,\alpha_r),$$

$$\varphi(n+1,\alpha_1,\ldots,\alpha_r) =$$

$$\beta(n,\alpha_1,\ldots,\alpha_r,$$
$$\varphi(n,\gamma_{11}(n,\alpha_1,\ldots,\alpha_r),\ldots,\gamma_{1r}(n,\alpha_1,\ldots,\alpha_r)),$$
$$\varphi(n,\gamma_{21}(n,\alpha_1,\ldots,\alpha_r),\ldots,\gamma_{2r}(n,\alpha_1,\ldots,\alpha_r)),$$
$$\ldots,$$
$$\varphi(n,\gamma_{k1}(n,\alpha_1,\ldots,\alpha_r),\ldots,\gamma_{kr}(n,\alpha_1,\ldots,\alpha_r)))$$

of a function with arbitrarily many argument places does not lead out from the class of primitive recursive functions."

Peter's result for the two argument case is easily seen, since it is just the theorem:

$$f(X_0,Y_0) = f'(X_0,Y_0),$$

justified above. This theorem can actually be proved completely automatically by the modified theorem prover described in [4]. Peter's proof of the theorem is somewhat complicated because she carries it out in number theory where a Goedel enumeration method must be used to express the notion of the list of x's used in the computation of f[4].

As noted in the introduction, it is possible to avoid the translation of the partial recursive specifications into primitive recursive ones and still prove many theorems. Moore [4] describes how. Roughly stated, Moore's approach requires two enrichments of the original theorem prover. First, the induction principle must be strengthened so that to prove $\varphi(X,Y)$ for all X and Y, where Y is used as an accumulator in some function f in φ, one first proves $\varphi(0,Y)$ for all Y, and then inductively assumes $\varphi(X,e(X+1,Y))$ for any expression e, and proves $\varphi(X+1,Y)$. Moore explains why this is a valid induction principle (also cf. Goodstein [2], pp. 123). The choice of the expression e is left to the theorem proving process, and Moore explains how it can be determined from the definition of f. The second augmentation required in [4] is the extension of the generalization heuristic so that accumulator argument positions which initially contain constants can be replaced by expressions containing free variables, allowing the use of the induction method above. This generalization introduces a new function, called the "accumulator function", into the accumulator positions. It turns out that Moore's accumulator function is just our finaly.

The method of translation into primitive recursive form proposed in this paper does not require either the enriched form of induction or the subtle generalization.

We have found that proofs of many theorems involving f' are complicated by the introduction of terms such as finaly(append(x,y),z), due to the expansion of the rev call in f'. However, use of the lemma:

```
finaly(append(x,y),z)
=
finaly(x,finaly(y,z)),
```

(which can be proved by the theorem prover) allows these terms to be further simplified.

In fact, the use of this equality usually yields the same lemma produced by accumulator generalization and induction in [4].

[4] This is a good example of why future number theory courses should be taught using Pure LISP as the meta-language.

REFERENCES

[1] Boyer, R.S. and Moore, J Strother. Proving theorems about LISP functions. *J. ACM 22*, 1 (January 1975), pp. 83-105.

[2] Goodstein, R. L. Studies in logic. North-Holland Publishing Company, Amsterdam, 1964.

[3] McCarthy, J. Recursive functions of symbolic functions and their computation by machine. *C. ACM*, 3 (April 1960).

[4] Moore, J Strother. Introducing iteration into the Pure LISP Theorem Prover. *IEEE Transactions on Software Engineering*, SE-1, No. 3 (September 1975), pp. 328-338.

[5] Peter, R. Recursive functions. Academic Press, New York, 1967, pp. 63-69.

Appendix C

PROJECT ACTIVITIES

Appendix C

PROJECT ACTIVITIES

A number of outside activities were undertaken on project. These activities consisted of attendance at conferences or workshops related to this project (in many cases presentations were made at these gatherings), and publication of papers covering, or closely related to the project work. The essential features of such activities are summarized below.

1. Invitational Workshops

* Workshop on "The Feasibility of Software Certification", held at Stanford Research Institute, Menlo Park, California on 14-15 November 1974.

A formal presentation, entitled "On Program Reliability and Specification" was given by J. M. Spitzen of our project staff.

* U. S. Army Computing Systems Command Workshop on Reliable Software, Falls Church, Virginia on 19-20 November 1974.

The SRI Computer Science Laboratory attendees were B. Elspas and Lawrence Robinson. (The latter individual was not a

member of this project team; Elspas' attendance was supported by this contract.) Elspas presented a talk describing the present state of the art of program verification and some predictions for future progress in this field. A digest of this talk (prepared by P. Fisher, W. Hankley, and J. McCall) appears in "Steps Toward Reliable Software: Proceedings of a Workshop," Technical Documentary Report, U. S. Army Computer Systems Command, pp. 3-30 through 3-46 (January 1975).

* Air Force-NASA Workshop on Fault-Tolerant Systems, held at Research Triangle Park, North Carolina on 3-5 December 1975.

B. Elspas presented a tutorial on program verification, also covering research progress at SRI on our current Air Force, Army, Office of Naval Research and NSF contracts concerned with program verification. An abstract of his talk appears in the Workshop Proceedings (Proceedings From the Fault Tolerant Systems Workshop, December 3-5, 1975, Research Triangle Institute, Box 12194, Research

Triangle Park, North Carolina 27709).

The Computer Science Laboratory attendees were B. Elspas and J. Goldberg. Mr. Goldberg's participation was supported by sources other than this contract.

B. Conferences Attended On This Project

ACM-74 Conference at San Diego, California was attended on 11-13 November 1974 by B. Elspas. No presentation was made by him.

ACM Symposium on Principles of Programming Languages at Palo Alto, California was attended on 20-22 January 1975 by B. Elspas, J. Spitzen, R. Shostak, and R. S. Boyer. No presentations were given by these project staff members.

International Conference on Reliable Software, held at Los Angeles, California was attended by B. Elspas on 21-23 April 1975. A paper entitled, "SELECT--A Formal System for Testing and Debugging Programs by Symbolic Execution," by R. S. Boyer, B. Elspas, and K. N. Levitt was delivered by B. Elspas. The subject matter of this paper concerned an experimental system for semiautomatic testing of programs for correctness which had previously been developed at SRI

under NSF Grant GJ-36903X. Preparation of the paper was covered by the NSF grant. However, attendance of Drs. Elspas and Boyer at the conference was supported by this AFOSR contract. The paper appears in full in the conference proceedings.

National Computer Conference (NCC 1975) held at Anaheim, California on 19-22 May 1975 was attended on 21 May by B. Elspas for the purpose of participating in a Panel Discussion on "Program Verification in 1980" (chaired by R. L. London). No written version of this talk is available.

Invitational DoD/Industry Conference on Software Verification and Validation, 3-5 August 1976, Syracuse, New York. Drs. B. Elspas and J. M. Spitzen attended this RADC/ARPA conference. They made two separate presentations. Elspas presented a paper co-authored with Spitzen, entitled "A System for the Verification of JOCIT Programs." This paper describes work under contract with RADC that makes use of techniques developed under the present AFOSR contract. A written version appears in the proceedings issued by RADC and ARPA. Spitzen's presentation was a survey of current work at SRI on the hierarchical development methodology (HDM), with particular reference to the design of a provably secure operating system. The paper appearing in the proceedings is an early version of the Spitzen, Levitt,

Robinson paper reproduced as Appendix A), but the oral presentation was, as indicated, broader in scope. The attendance of Drs. Elspas and Spitzen at this conference was supported under RADC Contract F30602-76-C-0204.

National Computer Conference (NCC-77) at Dallas, Texas, 13 June 1977. B. Elspas attended the NCC-77 in order to participate in a Panel Discussion on "Symbolic Execution" under the chairmanship of E. F. Miller, Jr. Dr. Elspas' attendance there was partially supported by this contract. No written copy of his remarks is available.

Fifth International Joint Conference on Artificial Intelligence - 1977 (IJCAI-77), Massachusetts Institute of Technology, Cambridge, Massachusetts, 22-25 August 1977. Robert E. Shostak presented his paper, "An Algorithm for Reasoning About Equality," covering portions of his research on the decision algorithm for Presburger arithmetic carried out under this AFOSR contract. His attendance at IJCAI-77 was supported in part by this contract.

AFOSR/ARO/ONR Conference on Research Directions in Software Technology, Providence, Rhode Island, 10-12 October 1977. B. Elspas attended this conference and participated by submitting (at the invitation of Prof. Jack Dennis, an Associate Chairman) a "Discussant Contribution" with respect

to one of the formal papers, "Progam Verification," by Ralph L. London. For unexplained reasons this contribution (along with some others) failed to be included in the Proceedings. Its inclusion is promised, however, in a companion book, Research Directions in Software Technology now being prepared by P. Wegner and several associate editors.

C. Papers Written Wholly or Partly on this Project

J. M. Spitzen, K. N. Levitt, and L. Robinson, "An Example of Hierarchical Design and Proof," appeared originally as Technical Report Number 2, SRI Project 4079 (March 1976), and was then submitted to C. ACM, Programming Languages Department; it has since been resubmitted after revisions prompted by referees' comments.

The preparation of this paper was jointly supported by the present AFOSR contract, NSF Grant DCR74-18661, and ONR Contract N00014-75-C-0816.

B. Wegbreit and J. M. Spitzen, "Proving Properties of Complex Data Structures," J. ACM, Vol. 23, No. 2, pp. 389-396 (April 1976).

Dr. Spitzen's contribution to the writing of this paper was supported by the present AFOSR contract; Dr. Wegbreit's contribution by Xerox Palo Alto Research Center.

R. S. Boyer, J S. Moore, and R. E. Shostak, "Primitive Recursive Program Transformation," presented by Dr. Shostak at the Third ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, held in Atlanta, Georgia on 19-21 January 1976.

Dr. Shostak's contribution to this paper was supported by the present AFOSR contract. Dr. Boyer's work was supported by ONR, and Dr. Moore by Xerox Palo Alto Research Center, where he was a staff member just before he joined the SRI Computer Science Laboratory.

Robert E. Shostak, "An Algorithm for Reasoning About Equality," Proc. IJCAI-77 Conference, Vol. 1, pp. 526-527 (August 1977). Submitted to J. ACM for publication.

Robert S. Boyer and J Strother Moore, "A Lemma Driven Automatic Theorem Prover for Recursive Function Theory," Proc. IJCAI-77 Conference, Vol. 1, pp. 511-519 (August 1977). Support for the preparation of this paper was shared

by the present AFOSR contract, ONR Contract N00014-75-C-0816
and NSF Grant DCR72-0373A01.