INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in “sectioning” the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again — beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from “photographs” if essential to the understanding of the dissertation. Silver prints of “photographs” may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

University Microfilms International
300 North Zeeb Road
Ann Arbor, Michigan 48106 USA
St. John’s Road, Tyler’s Green
High Wycombe, Bucks, England HP10 8HR
BRITTON, Dianne Ellen, 1950-
INCREMENTAL SYNTHESIS OF INDUCTIVE ASSERTIONS
FOR PROGRAM VERIFICATION.

The University of Arizona, Ph.D., 1977
Computer Science

University Microfilms International, Ann Arbor, Michigan 48106
INCREMENTAL SYNTHESIS OF INDUCTIVE ASSERTIONS
FOR PROGRAM VERIFICATION

by
Dianne Ellen Britton

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF COMPUTER SCIENCE
In Partial Fulfillment of the Requirements
For the Degree of
DOCTOR OF PHILOSOPHY
In the Graduate College
THE UNIVERSITY OF ARIZONA

1977
I hereby recommend that this dissertation prepared under my
direction by Dianne Ellen Britton
entitled Incremental Synthesis of Inductive Assertions
for Program Verification
be accepted as fulfilling the dissertation requirement for the
degree of Doctor of Philosophy

[Signature]
Dissertation Director  
November 17, 1977  

As members of the Final Examination Committee, we certify
that we have read this dissertation and agree that it may be
presented for final defense.

[Signature]  
[Signature]  
November 17, 1977  

Final approval and acceptance of this dissertation is contingent
on the candidate's adequate performance and defense thereof at the
final oral examination.
STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under the rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: [Signature]

[Signature]
ACKNOWLEDGMENTS

The programming and the preparation of this document were done at The University of Arizona Computer Center. I am grateful to the Department of Computer Science for funding me with respect to computer resources.

I would like to express my sincere thanks to my dissertation director, Richard J. Orgass. His enthusiasm for this work and his insights in times of technical difficulties helped provide the encouragement I needed to complete this research. Also, the weekly discussions with him and Ralph B. McLaughlin, although tangential to this work, have been of considerable value to me.

The efforts of Ralph E. Griswold and Larry H. Reeker, who served on my committee, are especially appreciated. Their careful reading and suggestions for improving this document have been invaluable.

Other persons to whom I am indebted for moral support and encouragement are Douglas K. Brotz, David R. Hanson, John T. Korb, Dolores M. Marik, and other members of the Department of Computer Science at The University of Arizona.

Finally, I am deeply grateful to Fred, who has given me boundless encouragement, understanding, and patience, and
who rearranged his life while I struggled through this endeavor. No words can truly express my appreciation.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF ILLUSTRATIONS</td>
<td>viii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>ix</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>Program Verification</td>
<td>2</td>
</tr>
<tr>
<td>In Defense of Verification</td>
<td>4</td>
</tr>
<tr>
<td>The Method of Inductive Assertions</td>
<td>8</td>
</tr>
<tr>
<td>Program Verifiers</td>
<td>11</td>
</tr>
<tr>
<td>This Research</td>
<td>16</td>
</tr>
<tr>
<td>Earlier Work in Proof Partitioning</td>
<td>17</td>
</tr>
<tr>
<td>Earlier Work in Assertion Synthesis</td>
<td>18</td>
</tr>
<tr>
<td>An Incremental Approach</td>
<td>21</td>
</tr>
<tr>
<td>Developing Inductive Assertions</td>
<td>25</td>
</tr>
<tr>
<td>Overview of This Dissertation</td>
<td>27</td>
</tr>
<tr>
<td>2. AN AXIOMATIC FRAMEWORK</td>
<td>29</td>
</tr>
<tr>
<td>Axiomatic Semantics</td>
<td>31</td>
</tr>
<tr>
<td>Formula Syntax</td>
<td>31</td>
</tr>
<tr>
<td>Formula Semantics</td>
<td>32</td>
</tr>
<tr>
<td>Proof Theory</td>
<td>33</td>
</tr>
<tr>
<td>A Clarification</td>
<td>33</td>
</tr>
<tr>
<td>Assertions</td>
<td>34</td>
</tr>
<tr>
<td>Input Conditions</td>
<td>35</td>
</tr>
<tr>
<td>Imbedding Assertions in Programs</td>
<td>36</td>
</tr>
<tr>
<td>Lemmas</td>
<td>39</td>
</tr>
<tr>
<td>3. DEVELOPING INDUCTIVE ASSERTIONS</td>
<td>44</td>
</tr>
<tr>
<td>Terminology</td>
<td>44</td>
</tr>
<tr>
<td>Exit Assertions</td>
<td>45</td>
</tr>
<tr>
<td>Non-Inductive Invariants</td>
<td>49</td>
</tr>
<tr>
<td>Nested Loops</td>
<td>54</td>
</tr>
<tr>
<td>4. AN ALGORITHM FOR INCREMENTAL VERIFICATION</td>
<td>65</td>
</tr>
<tr>
<td>Weakest Preconditions</td>
<td>66</td>
</tr>
<tr>
<td>The Algorithm</td>
<td>68</td>
</tr>
<tr>
<td>The Chief Executive</td>
<td>69</td>
</tr>
<tr>
<td>Subordinates</td>
<td>70</td>
</tr>
</tbody>
</table>
### TABLE OF CONTENTS—Continued

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5. A PROTOTYPE INCREMENTAL VERIFICATION SYSTEM</td>
<td>78</td>
</tr>
<tr>
<td>System Overview</td>
<td>78</td>
</tr>
<tr>
<td>Trial Invariant Generator</td>
<td>81</td>
</tr>
<tr>
<td>Theorem Prover</td>
<td>82</td>
</tr>
<tr>
<td>An Interactive Formula Simplifier</td>
<td>85</td>
</tr>
<tr>
<td>Conditional Evaluator</td>
<td>86</td>
</tr>
<tr>
<td>User Interface</td>
<td>88</td>
</tr>
<tr>
<td>Example</td>
<td>91</td>
</tr>
<tr>
<td>Discussion</td>
<td>93</td>
</tr>
<tr>
<td>Factorial Example</td>
<td>95</td>
</tr>
<tr>
<td>Insights from Experience</td>
<td>99</td>
</tr>
<tr>
<td>Recursion and Trial Invariants</td>
<td>100</td>
</tr>
<tr>
<td>Expressing Properties of Loops</td>
<td>105</td>
</tr>
<tr>
<td>Summary</td>
<td>108</td>
</tr>
<tr>
<td>6. CONCLUSIONS</td>
<td>109</td>
</tr>
<tr>
<td>Summary</td>
<td>109</td>
</tr>
<tr>
<td>Contributions</td>
<td>111</td>
</tr>
<tr>
<td>Incremental Nature of the Proof Method</td>
<td>111</td>
</tr>
<tr>
<td>Step-Wise Development of Inductive Assertions</td>
<td>114</td>
</tr>
<tr>
<td>Nested Loops</td>
<td>116</td>
</tr>
<tr>
<td>Human-Machine Interface</td>
<td>117</td>
</tr>
<tr>
<td>The Backtracking Verification Algorithm</td>
<td>119</td>
</tr>
<tr>
<td>Directions for Future Research</td>
<td>119</td>
</tr>
<tr>
<td>Implementation of Heuristic Generation of Invariants</td>
<td>120</td>
</tr>
<tr>
<td>Toward a Realistic Verification System</td>
<td>121</td>
</tr>
<tr>
<td>Orderly Generation of Trial Invariants</td>
<td>123</td>
</tr>
<tr>
<td>Exploiting Lemmas</td>
<td>124</td>
</tr>
<tr>
<td>Experimentation</td>
<td>125</td>
</tr>
<tr>
<td>Incorrect Programs</td>
<td>125</td>
</tr>
<tr>
<td>Perspective</td>
<td>126</td>
</tr>
<tr>
<td>APPENDIX A: SYNTAX AND AXIOMATIC SEMANTICS FOR L</td>
<td>127</td>
</tr>
<tr>
<td>Syntax</td>
<td>127</td>
</tr>
<tr>
<td>Axiomatic Semantics</td>
<td>129</td>
</tr>
<tr>
<td>Formulas</td>
<td>129</td>
</tr>
<tr>
<td>Semantics</td>
<td>129</td>
</tr>
<tr>
<td>Proof Theory</td>
<td>129</td>
</tr>
</tbody>
</table>
### TABLE OF CONTENTS—Continued

<table>
<thead>
<tr>
<th>Appendix</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>DESCRIPTION OF THE ASSERTION LOGIC AL</td>
<td>134</td>
</tr>
<tr>
<td></td>
<td>Syntax</td>
<td>134</td>
</tr>
<tr>
<td></td>
<td>Semantics</td>
<td>135</td>
</tr>
<tr>
<td></td>
<td>Notation</td>
<td>135</td>
</tr>
<tr>
<td></td>
<td>The Interpretation Function</td>
<td>136</td>
</tr>
<tr>
<td></td>
<td>Proof Theory</td>
<td>139</td>
</tr>
<tr>
<td></td>
<td>Notation and Terminology</td>
<td>139</td>
</tr>
<tr>
<td></td>
<td>Proofs</td>
<td>139</td>
</tr>
<tr>
<td></td>
<td>Rewrite Rules</td>
<td>140</td>
</tr>
<tr>
<td>C</td>
<td>PROOF OF THE WEAKEST PRECONDITION</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>Function for L</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>Derived Rules of Inference for L</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>The Proof</td>
<td>144</td>
</tr>
<tr>
<td>D</td>
<td>INCREMENTAL VERIFICATION ALGORITHM</td>
<td>149</td>
</tr>
<tr>
<td></td>
<td>For L</td>
<td>149</td>
</tr>
<tr>
<td></td>
<td>Chief Executive</td>
<td>149</td>
</tr>
<tr>
<td></td>
<td>Subordinates</td>
<td>149</td>
</tr>
<tr>
<td>E</td>
<td>JUSTIFICATION OF THE TREATMENT OF AXIOMS BY IFS</td>
<td>154</td>
</tr>
<tr>
<td></td>
<td>Notation</td>
<td>155</td>
</tr>
<tr>
<td></td>
<td>Justifications</td>
<td>155</td>
</tr>
<tr>
<td>F</td>
<td>EXAMPLE PROGRAMS AND TRANSCRIPTS</td>
<td>159</td>
</tr>
<tr>
<td></td>
<td>User-Defined Predicates and Functions</td>
<td>160</td>
</tr>
<tr>
<td></td>
<td>Binary Search Program</td>
<td>161</td>
</tr>
<tr>
<td></td>
<td>Selection Sort Program</td>
<td>166</td>
</tr>
<tr>
<td></td>
<td>Minimal Spanning Tree Program</td>
<td>176</td>
</tr>
<tr>
<td></td>
<td>REFERENCES</td>
<td>180</td>
</tr>
</tbody>
</table>
**LIST OF ILLUSTRATIONS**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Flow Diagrams for Control Structures</td>
<td>71</td>
</tr>
<tr>
<td>2. Organization of the Prototype Verification System</td>
<td>79</td>
</tr>
</tbody>
</table>
ABSTRACT

A severe drawback to using the method of inductive assertions to verify programs is the difficulty of discovering the necessary inductive assertions. Empirical observation shows, however, that the inductive assertions required to verify a program are usually conjunctions of individual loop invariants. This dissertation shows how each of these loop invariants can be actively sought by considering some constraints on the inductive assertion, e.g., it must be inductive and it must be strong enough to verify the loop's exit condition.

An incremental approach to verification is described which permits inductive assertions to be developed step-by-step by finding the individual loop invariants one at a time and which effectively partitions a program's proof of correctness into a number of smaller, simpler proofs. The approach is called "incremental" because a typical verification consists of several successful passes through a program verifier. The user initiates each pass by augmenting the program with new assertions expressing additional properties of the program. Each successful pass validates these user-specified assertions and possibly some additional inductive assertions as well. Assertions that have been validated in
previous passes are recognized in subsequent passes as facts, or lemmas, which can be drawn upon to synthesize and/or validate other assertions. During any one pass, if the inductive assertions supplied with the program are not sufficient to complete the pass, new inductive assertions are generated as they are needed.

Verifying a program incrementally can be used to gradually develop the inductive assertions necessary to verify a program: each pass through the verifier finds loop invariants one at a time and conjoins them to the program's loop assertions. As a result, the incremental approach provides a way of dealing with the complexity of developing inductive assertions for nested loops. Since a program is correct only if each of its loops is correct, the incremental approach can be used to verify that each loop individually does what it is expected to do. This dissertation shows that if inner loops are processed before outer ones, then the inductive assertions for all loops can be built up step-by-step over several passes as more loops are verified.

A simple Algol-like programming language L is defined, which supports incremental verification. In L, program properties are expressed with assert and lemma clauses. An axiomatic semantic description of L is given, which defines assertions as formulas to be proved and lemmas as formulas to be assumed during verification.
An algorithm for incrementally verifying programs in L is given. The algorithm, which processes programs in a top-down fashion, is based on the computation of weakest preconditions. Whenever an inductive assertion is found to be inadequate, the algorithm backtracks, uses any available methods to generate an alternative inductive assertion, and continues with the verification. Thus, in addition to supporting a multi-pass approach to verification, the algorithm provides a means for controlling the generation and testing of alternate inductive assertions, such as is required by heuristic methods for automatic assertion synthesis.

A verifier that supports incremental verification for programs written in L has been implemented in Simula 67. In order to generate alternative inductive assertions, this verifier simply requests them from the user. Theorem proving is done with an interactive formula simplifier that supports user-defined predicates. Using this verifier, inductive assertions have been developed for a number of small programs, including a binary search program and a selection sort.
CHAPTER 1

INTRODUCTION

At one time, virtually the only available method of ascertaining the correctness of a program was to subject it to intensive reading, testing, and debugging. In fact, these methods are still used nearly exclusively. Somewhere in this informal process, however, it becomes necessary to make an inductive leap: if the program has worked correctly for a particular set of inputs, then it will work correctly for all subsequent sets of allowable inputs. Unfortunately, determining correctness in this manner places too much responsibility on human intuition. Evidence of this is provided by the persistence of errors in thoroughly read, tested, and "debugged" programs.

With the appearance of Floyd's paper "Assigning Meanings to Programs" in 1967, a theory for formally determining the correctness of programs was presented in sufficient detail to encourage research in the area now referred to as "program verification". In a formal proof of correctness, each step follows logically from the next by virtue of an axiom or rule of inference in some formal system of logic. Because program verification provides formal proofs
for programs, the reliance on human intuition for the correctness of the proof is minimized.

**Program Verification**

The objective of program verification is to provide formal mathematical proofs that programs meet their specifications. If a program and its specifications cannot be proved consistent, then the verification process should also help isolate the error.

In order for program verification to be applicable, the program's specifications and the semantics of the programming language in which the program is written must be formally, or at least semi-formally, expressed. Since proofs of even relatively shallow theorems in any formal system of logic are often lengthy and tedious, it is not surprising that proofs of correctness for less than trivial programs are likewise lengthy and tedious, and tremendously error-prone if constructed manually. Thus, although verification methods can be applied without any assistance from a computer, if program verification is ever to be successful as a means of attaining reliable software, it must be at least partially automated.

A variety of inductive methods for proving properties of programs have been developed, each of which have favored domains of applicability. One of the earliest
methods was recursion induction (McCarthy 1963), which has been used mainly to prove equivalence between programs. Computational induction (Park 1969) deals mainly with recursive programs and is the basic induction rule for Milner's LCF proof-checker (Milner 1972; Scott 1969). Structural induction (Burstall 1969) is extremely useful in proving properties of recursive programs that operate on recursive data structures; structural induction is the induction rule used by the LISP verifiers (Boyer and Moore 1975; Cartwright 1976). Datatype induction (Hoare 1972; Spitzen and Wegbreit 1975; Wegbreit and Spitzen 1976) applies to datatype-defining modules such as Simula 67 classes, CLU clusters, or Alphard forms. The method of inductive assertions (Naur 1966; Floyd 1967) is the principal one used for iterative programs with assignments, and it can be combined with other induction rules to extend its applicability to recursive functions. Subgoal induction (Morris and Wegbreit 1977) is applicable to recursive and iterative programs. It is a symmetric alternative to the method of inductive assertions, and the two methods may be used to supplement each other. Whereas the method of inductive assertions reasons "forward" from the start of the computation, subgoal induction reasons "backward" from the end of the computation. Continuation induction (Topor 1975) is also suitable for recursive and iterative programs,
including those with arbitrary jumps. This method, however, usually yields more difficult proofs for iterative programs than the method of inductive assertions. The work in this dissertation, which is concerned with iterative programs, lies entirely within the context of inductive assertions.

In Defense of Verification

Program verification has been attacked on some fundamental grounds. Before continuing, it perhaps is advisable to address these objections.

DeMillo, Lipton and Perlis (1977) are convinced that proofs of programs will not be believed because they are not subject to the same kinds of social processes that lend credence to mathematical theorems. These social processes are reading, refereeing, discussion, internalization and paraphrasing, generalization, use, and connection with other theorems. Since, they argue, the purpose of program verification is to construct a proof of a program which will convince people that the program is correct, if people are not convinced, then the verification has accomplished nothing and the effort has been wasted.

On the contrary, program proofs and theorems about programs have the potential of being subject to the very same social processes to which mathematical theorems are subject. The fact that this has not yet occurred is only a
reflection on the state of the art of programming and program verification. In support of this claim, consider the relationship between abstraction, which is another line of research being pursued for the purpose of obtaining reliable software (Wulf, London and Shaw 1976; Liskov et al. 1977; Hoare 1972; Guttag 1975; Guttag, Horowitz and Musser 1976), and program verification.

Whereas verification is usually viewed as an a posteriori means of determining the correctness of programs that are already written, abstraction is more closely tied with the a priori construction of correct programs by means of high-level, top-down design. Abstraction can be a useful tool for program reliability because it encourages program modularity, exporting of program modules to other programs, and hiding of implementation details where those details are not of interest or are not to be known. When abstractions are identified with formal specifications for programs, abstractions generate the following possibilities. First, programs become subject to formal verification. Second, and more importantly, abstractions can be used in place of the programs they specify during the design and verification of other programs. Assuming that, in the quest for reliable software, algorithms and programs will be recycled much more in the future than at present, it can be expected that they will be subject to those social processes described by DeMillo et al. for mathematical theorems. Because of the
second possibility stated above, theorems about programs and program proofs will be as subject to these social processes as the programs themselves.

A related objection brought up by DeMillo et al. is that program proofs will not be read, that is, scrutinized carefully by humans, because the proved theorems are complicated and uninteresting and the proofs are exceedingly lengthy and detailed. This objection is unwarranted. The theorems proved during program verification fall into two classes: those theorems about the program, which are as interesting as the program they are about and which are able to be simply expressed for the bulk of interesting programs, and verification conditions, i.e., lemmas whose truth implies the correctness of the program. Verification conditions are often complicated and uninteresting, but there is no reason that a reader need ever see them. Although automation of the verification process would eliminate the need to error-check program proofs entirely, it is generally agreed that total automation is, if not impossible, then computationally infeasible. Nevertheless, the process can certainly be partially automated, with the user filling in the remaining details using informal reasoning as necessary. In this case, the informal, user-supplied steps are the only steps of the proof that need be read and error-checked.

Moreover, these steps should be sufficiently interesting and
few that a program's "proof" will be well fit for human consumption. It is to be expected that as experience is gained with program proof paradigms that informal ideas will gradually become formalized, so that the number of informalities introduced in a program proof remains small.

Another objection that has been raised against program verification is that proofs of really important programs are necessarily too complex to be computationally feasible. It is undoubtedly true that there are many programs in existence whose verification would be computationally intractable (even if the programs were actually correct). Nevertheless, it is unfair to extrapolate from this that programs to solve significant problems cannot be practically verified. Program verification and program design really should be considered hand-in-hand. (Recall the brief discussion of abstraction and program verification, above.) The same modularization that keeps a large program conceptually under control also keeps the complexity of a program's proof under control. Thus, as the art of program design matures, it can be expected that more and more important programs will be successfully verified. Already, some progress has been made toward verifying significant programs: a program to generate verification conditions (Ragland 1973) has been formally verified. This program consists of 203 procedures, each of which is less than a page in length.
The verification conditions were generated by an automatic verification condition generator (Wang 1973). Although the verification conditions were all proved manually, it is believed that about 70 percent of them are simple enough to be proved automatically.

The Method of Inductive Assertions

Using the method of inductive assertions, a number of assertions are attached to the program to be verified. Assertions are formulas in a formal logic such as the first-order predicate calculus. An assertion provides specifications for a program by stating what relationships between program variables are expected to be true at the point where the assertion is attached to the program. The effect of a program's assertions is to partition the program into a finite number of paths, with an assertion at the beginning and end of each path.

Taking into account the semantics of the programming language, each path with its starting and ending assertions determines a verification condition, which is another formula. Verification conditions may be generated by symbolically executing the path (Deutsch 1969; Topor 1975; King 1976), by propagating the ending assertion backwards over the path to the starting assertion (Floyd 1967; King 1969), or by appealing to an axiomatic definition of the
programming language (Igarashi, London and Luckham 1975). If all the verification conditions can be proved to be true, then the program is consistent with its specifications, i.e., its assertions. Provided that the assertions at the start and end of the program adequately express the intended effect of the program, consistency of the program with its specifications implies (partial) correctness. ("Partial" means that the question of termination of the program has not been addressed.)

The method of inductive assertions requires that the paths determined by the attached assertions be finite. To meet this requirement, it is necessary and sufficient that at least one assertion be attached to each loop (Floyd 1967). An assertion attached to a loop is an inductive assertion, so called because it expresses a property of the program that is expected to be true on each traversal of the loop. A serious drawback of this method is that the detailed inductive assertions that are necessary to verify a program are often less than intuitive and are not easily constructed by the programmer. In hopes of improving this situation, some attention has recently been given to developing methods by which inductive assertions can be generated mechanically. However, since assertion synthesis is known to be at least as hard as any NP-complete problem (Wegbreit 1976), nobody foresees a practical verification system in
which the required inductive assertions are synthesized entirely automatically.

Following the terminology of Katz and Manna (1976), two general approaches to assertion synthesis can be distinguished:

1. **Algorithmic** approaches synthesize inductive assertions by analyzing the program to determine what it does. These inductive assertions (loop invariants) are guaranteed to be valid for the program.

2. **Heuristic** approaches synthesize inductive assertions using the program's specifications. The validity of these inductive assertions (trial loop invariants) need to be ascertained through verification.

Both of these approaches, however, perform rather poorly for nested loops. Algorithmic methods must tolerate a substantial loss of information during analysis in order to keep the computation within reasonable time and space bounds, and nested loops tend to aggravate this information loss. Heuristic methods rapidly lose control of the interdependencies that develop among the inductive assertions of nested loops. This dissertation addresses itself chiefly to the problem of synthesizing inductive assertions.
Program Verifiers

Much of the research in program verification has been focused through attempts to implement automatic and semi-automatic program verifiers. The first automatic verifier was King's (1969). As a first approximation of an automatic verification system, King's verifier must be considered highly successful. Nevertheless, it quickly became evident that in order to become practical, program verifiers would have to be made more sophisticated in a number of areas.

Theorem proving. From the start, a fertile area of improvement for program verifiers has been the theorem proving component. The difficulty associated with theorem proving has been attacked in a number of different ways, one of which is to incorporate human interaction into the theorem proving process. In such systems, theorem proving is usually based on natural deduction proof methods, so that the user can work in a reasonably intuitive environment. Examples of verification systems with interactive, natural deduction theorem provers are PIVOT (Deutsch 1969), the interactive verification system of Good, London and Bledsoe (1975) which uses the interactive theorem prover of Bledsoe and Bruell (1973), Cartwright's verifier for TYPED LISP (Cartwright 1976), and Topor's POP-2 verifier (Topor 1975).
Another improvement in the theorem proving component of verification systems has been to augment or replace the theorem prover with a simplifier. In this case, a simplifier reduces a verification condition to a simpler form before passing it to the theorem prover. If the verification condition simplifies to TRUE, then the theorem prover need not be invoked at all. Some verifiers that use simplifiers are PIVOT, the Stanford University Pascal verifier (Igarashi et al. 1975; Suzuki 1975; VonHenke and Luckham 1974), and the University of California Information Sciences Institute (ISI) verifier for algebraic datatypes (Guttag et al. 1976).

Expressibility. In order to be practical, a program verifier must provide facilities for improving the expressibility of assertions and theorems about programs. Thus, another desirable property of theorem provers to be used for verification is that they be extendable to theorems in many different domains. Research in this direction has been pursued at ISI (Waldinger and Levitt 1974). The ISI theorem prover, which is written in QA4 LISP, takes advantage of the data structures available in QA4, e.g., bags and tuples, to simplify theorem proving strategies. The theorem prover can be easily extended by adding functions that implement new predicates, proof strategies and axioms for a particular domain.
Another way of defining new predicates is through macro-like definitions which are input with the program being verified. This approach is supported by PIVOT and the Stanford Pascal verifier (Suzuki 1975). The Stanford Pascal verifier also allows the user to define new predicates implicitly by submitting a file of axioms to the system's formula simplifier. The axioms are then translated into reduction rules and subgoal strategies to be used during simplification. Similarly, the theorem provers used in the Good, London and Bledsoe verifier and in Topor's verifier accept user-supplied axioms, but interactively.

A rather different tactic for improving expressibility has been taken by the LISP verifiers (Boyer and Moore 1975; Boyer and Moore 1977; Cartwright 1976). In these verifiers, the specification language is the same as the programming language. For the Boyer and Moore theorem prover this is a modified subset of Pure LISP, and for the Cartwright theorem prover it is TYPED LISP, another subset of Pure LISP augmented with a recursive type definition facility. Similarly, the ISI verifier for algebraic data-types (Guttag et al. 1976) expresses programs and axioms in the same language. In fact, programs consist mostly of axioms, which are used as rewrite rules for theorem proving.

In Topor's verifier, the effect of a program or program part is expressed with a virtual program. Virtual programs are syntactically and semantically just like actual
programs, except that virtual programs must not contain loops but can include references to user-defined predicates and functions.

**Constructing inductive assertions.** As noted earlier, a special problem with the method of inductive assertions is that it requires an inductive assertion for each program loop. Two verification systems that construct inductive assertions automatically are an ISI verification system (Elspas 1974) and VISTA (German and Wegbreit 1975). The ISI system does not attempt to generate all the inductive assertions needed to prove correctness, but only some that reflect the program's actual behavior. This system works by constructing from the program's loops a set of recurrence relations, which are solved interactively with the help of the user. VISTA uses both algorithmic and heuristic methods to synthesize assertions. These include weak interpretation, forward propagation of valid assertions, combining output assertions with loop exit assertions and generalizing as necessary, and analyzing proofs that fail. VISTA has generated all the inductive assertions to verify King's first seven examples, as well as approximately half of those needed to verify the program FIND (Hoare 1971).

**Interaction.** As has already been described, interaction has been used in program verifiers as a practical means of extending the power of theorem proving, for
instance, by guiding the proof process or supplying additional axioms and assumptions. In addition, other uses of interaction are to be found among program verifiers.

1. PIVOT permits the user to interactively modify the program and/or its assertions when a verification condition cannot be proved. If the verification of a path remains intact after a program modification, PIVOT is clever enough to recognize that the path need not be re-verified.

2. The VISTA user is responsible for invoking the loosely coupled modules that comprise the verification system. In this case, interaction has been substituted for some forms of heuristic control that are not yet understood.

3. Cartwright's TYPED LISP verifier interactively accepts theorems and supporting lemmas from the user. Although the verifier remembers which theorems and lemmas have yet to be proved, the user is free to initiate their proofs in whatever order seems convenient.

4. In much the same manner that other verifiers depend on the user to aid in theorem proving, the SRI inductive assertion generator (Elspas 1974) depends on the user to aid in solving difference equations (recurrence relations). Solutions to these equations, which are derived
directly from the program text, yield loop invariants. The user helps by suggesting formula manipulations and strategies for solving the equations.

5. The ISI verifier for abstract datatypes (Guttag et al. 1976) lets the user choose the order in which verification conditions are proved. Also, whenever the proof of a verification condition requires an additional assumption, the system displays a reduced form of the verification condition to the user. The user may supply the required assumption, which later must be proved to be an invariant of the datatype or its implementation (using datatype induction). Otherwise, the proof proceeds entirely automatically.

**This Research**

This research introduces an incremental approach to program verification and the synthesis of inductive assertions, in which a program's proof of correctness is partitioned into a number of smaller, simpler proofs. This approach is not entirely automatic since it takes advantage of human insight and allows the programmer to loosely direct the search for inductive assertions. It is compatible with all algorithmic and most heuristic methods for assertion synthesis, in the sense that these other methods may be
combined with the incremental approach in order to more fully automate the process. In addition, this approach provides a way of dealing effectively with the problem of constructing inductive assertions for nested loops.

Earlier Work in Proof Partitioning

The possibility of partitioning the proof of correctness of a single program module into a number of simpler proofs seems to have been largely overlooked in the design of automatic verification systems. The usefulness of such an approach, however, was perceived very early in the history of program verification. In "Assigning Meanings to Programs" (Floyd 1967), the original defining paper for program verification, Floyd notes that for a semantic definition of a programming language to be satisfactory, the following must be true: Suppose separate proofs exist that a program has certain properties, so that each proof follows from a different set of assertions attached to the program. Then these proofs may be combined into one by forming the conjunction of the several assertions appearing at any single point in the program. (Here Floyd has loosely identified a proof with a set of valid assertions for a program.) Thus, for any program written in a programming language with a satisfactory semantic definition, in order to prove a complex property of a program, it is sufficient to
prove separately several simpler properties, whose conjunction is the complex property of interest.

The paper by Morris and Wegbreit on subgoal induction (1977) presents another method of proof partitioning, which more closely resembles the incremental approach to be described in this dissertation. Morris and Wegbreit suggest that the method of inductive assertions and subgoal induction can be combined in a two-pass approach to verify programs. In the first pass, the method of inductive assertions is used to validate a set of inductive assertions that are insufficient to completely verify the program. On the next pass, subgoal induction completes the verification, making use of the validated assertions from the first pass. Thus, their method divides a verification into separate passes and makes use of information that has been gained in one pass in order to complete the next pass.

Earlier Work in Assertion Synthesis

As mentioned earlier, approaches to assertion synthesis fall into two categories: algorithmic methods, which synthesize inductive assertions by analyzing the program to determine what it does, and heuristic methods, which synthesize inductive assertions using the program's specifications.

Algorithmic methods. Most of the algorithmic methods for synthesizing assertions are the so-called
"inside-out" methods (Elspas 1974; Green 1972; Caplain 1975; Katz and Manna 1973; Katz and Manna 1976), which find invariants for innermost loops first and systematically work their way to the outer loops. For the purpose of computing invariants for a loop that encloses another one, the inner loop is characterized by whatever invariants have been discovered for it. Methods for analyzing loops include using recurrence relations (Green 1972; Elspas 1974; Katz and Manna 1973; Katz and Manna 1976) and linear algebra (Caplain 1975).

Another algorithmic method is weak interpretation (Wegbreit 1975). In this method, the input assertion, which constrains the relations among the initial values of program variables, is propagated over the arcs of a program's flowgraph. Whenever a join point is reached in the flowgraph, the assertion placed on the outgoing arc is chosen to be weaker than the assertions on all incoming arcs. At the end of the interpretation, each arc has been tagged with an assertion that specifies (some of the) possible relations among program variables when the arc is traversed during program execution. Weak interpretation has been exploited by German and Wegbreit (1975) to deduce order relations among program variables, and the method is applicable to other simple properties as well.
An advantage of the algorithmic approaches to assertion synthesis is that valid assertions are generated independently of whether or not the program is "correct". In fact, it may be possible for a programmer to discover that a program is incorrect by inspecting the invariants generated for it. A disadvantage is that the generated invariants are seldom sufficient to verify the program; in order to keep the computation within reasonable time and space bounds, a substantial loss of information must be tolerated during analysis. Nested loops aggravate this problem for inside-out methods because the invariants generated for an outer loop are dependent on the invariants generated for inner loops.

Heuristic methods. Heuristic methods for synthesizing assertions (Wegbreit 1973; Wegbreit 1974; German and Wegbreit 1975; Greif and Waldinger 1974; Katz and Manna 1973) use the program's assertions to formulate trial invariants. For instance, the exit condition may be propagated backwards through the loop or combined in some manner with the loop entry condition to form a trial invariant. Since a program's specifications easily may be inconsistent with respect to what the program actually does, any trial invariants generated using a heuristic method for assertion synthesis need to be verified for validity. Verification of a trial invariant may necessitate that inductive assertions
be generated and verified for loops enclosed in the one for which a trial invariant has been generated. Moreover, output specifications for inner loops are usually not available until inductive assertions have been chosen for outer loops. For these reasons, heuristic methods often result in processing the program "outside-in", i.e., from outermost loops to inner ones.

Heuristic methods can sometimes generate loop invariants that algorithmic methods cannot, because the program's assertions may provide information that is quite difficult to extract from the program text alone. Another advantage of heuristic methods is that they generally do not expend effort generating loop invariants that are irrelevant with respect to verifying the program, because their efforts are constrained by the assertions in the program.

A disadvantage is that, if the program is incorrect, a program's assertions may be misleading, hindering rather than helping the search for loop invariants. Also, heuristic methods perform poorly for programs that contain nested loops, due to interdependencies among the inductive assertions.

An Incremental Approach

The approach to verification and assertion synthesis presented in this paper provides a natural and intuitive way
of partitioning a correctness proof for a program into a number of smaller, simpler proofs. The approach is "incremental" because a typical verification of a program consists of several successful passes through a program verifier. Each successful pass verifies another property of the program, and by doing so validates another set of assertions for the program. The novelty of this approach is that assertions that have been validated in previous passes are recognized in subsequent passes as facts, or lemmas, which can be drawn upon to synthesize and/or validate other assertions.

Each pass consists of the following steps:

1. First, the user augments the program with assertions by adding one or more assert clauses to the program text. Typically, the user adds one "interesting" assertion, which specifies a property of the program that the user believes to be true, and perhaps some inductive assertions as well.

2. The verifier takes the augmented program as input, generates verification conditions, and submits them, one by one as they are generated, to a theorem prover for proof.

3. An unprovable verification condition indicates that the inductive assertion for some loop is either missing or
inadequate, or that the loop is erroneous. Upon encountering an unprovable verification condition, the verifier backs up, synthesizes an alternate inductive assertion for the offending loop, and continues generating verification conditions (see step 2). Synthesizing the inductive assertion may be done through the use of heuristic methods or through interaction with the user.

4. When a complete set of verification conditions for the program has been generated and proved, the verifier outputs a new program. This program is exactly like the input program except that the assert clauses have been removed and replaced by lemma clauses that reflect the newly validated assertions.

If, at the end of a pass, the program's lemmas are not yet strong enough to imply to the user that the program is correct, the user initiates another pass. This time, the program to be augmented with assertions is the output program from the most recent pass.

The incremental approach has some important advantages over the conventional one-pass approach to verification:

1. The verification conditions generated in each pass tend to be shorter and/or easier to prove than those generated in a conventional verification system.
2. Due to the presence of lemmas in the program text, the inductive assertions required to complete each pass are substantially simpler than the inductive assertions needed in a one-pass approach.

3. Complex inductive assertions (actually, inductive lemmas) are built up gradually as more passes are completed.

4. In the case that the program is actually incorrect, this fact will surface during one of the passes, when a satisfactory inductive assertion cannot be found for some loop. Although correcting the program necessitates starting the verification over with the first pass, the particular pass in which the error is discovered helps isolate the error.

A verifier that supports incremental verification has been implemented. The entire system is written in Simula 67 and runs on the DECsystem-10. The design is sufficiently general to support a variety of iterative control structures, although only a few basic ones have been implemented. In order to generate alternative inductive assertions, this verifier uses the option of interacting with the user. The system has successfully used the incremental approach in order to synthesize inductive assertions for a
number of small programs, including a binary search program and a selection sort.

In order to support incremental verification, the source language recognized by this verifier has been enriched by permitting some statements in the language to contain optional **assert** and **lemma** clauses. A syntactic description and Hoare-style axiomatic semantics for these statements are provided in this dissertation. The advantage of using the program itself, rather than an external data base, for storing properties known to be true of the program is that it then becomes reasonable to describe (and use) valid assertions by appealing to the same formal semantics that is used to describe the programming language. Thus, the value of valid assertions can be exploited in a manner that is consistent with respect to the programming language as a whole and without resorting to ad-hoc mechanisms.

Developing Inductive Assertions

The inductive assertions required to verify a program are often surprisingly complex. This is true even for some short, simple programs. Nevertheless, the complexity of inductive assertions can be dealt with in an effective way, as suggested by the following empirical observation: most inductive assertions are merely conjunctions of individual loop invariants. Moreover, each loop invariant that
appears in an inductive assertion is there to ensure that either

1. The loop's exit assertion is provably true when the loop terminates, or

2. The loop's inductive assertion is provably inductive, i.e., it can be proved that if the assertion is true on one traversal of the loop, then it is again true on the next traversal.

In the case of nested loops, developing inductive assertions becomes very complicated due to interactions among the inductive assertions of the individual loops. In general, whenever an (inner) loop is enclosed in another (outer) loop, the inner loop's inductive assertion must be strong enough to verify that the inductive assertion of the outer loop is actually inductive. Additionally, whenever an (outer) loop encloses another (inner) loop, the outer loop's inductive assertion must be strong enough to verify that the inner loop's inductive assertion is true when the inner loop is entered.

The incremental approach provides a way of dealing with the complexity of developing inductive assertions for nested loops. Since a program is correct only if each of its loops is correct, the incremental approach can be used to verify that each loop individually does what it is
expected to do. This dissertation shows that if inner loops are processed before outer ones, then the inductive assertions for all loops can be built up step-by-step over several passes as more loops are verified.

Because the incremental approach is able to utilize lemmas that appear in a program text, it is easy to take advantage of algorithmic methods for assertion synthesis in an incremental verification environment. Algorithmic methods can be used to preprocess the program being verified, to insert mechanically-generated lemmas into the original text of the program to be verified. Additionally, this dissertation shows that, by combining heuristic assertion synthesis methods with incremental verification, the disadvantages associated with the heuristic methods can be largely overcome. The effect of an incremental approach is to impose additional structure on the assertion synthesis process by allowing the programmer to loosely direct the search for inductive assertions.

Overview of This Dissertation

This first chapter has given some background material for program verification, introduced an incremental approach to verification, which can be used to ease the difficulty of finding the inductive assertions needed to verify a program, and placed this work in relation to previous
research. In Chapter 2 a set of programming language constructs that support incremental verification, together with appropriate rules of inference for their axiomatic semantic description, are presented. Chapter 3 shows how inductive assertions can be developed incrementally, giving several examples. An algorithm is given in Chapter 4 for incremental verification, which backtracks to find alternative inductive assertions when the ones supplied with the program are inadequate. In Chapter 5 a prototype incremental verification system is described. This system proves verification conditions by means of an interactive formula simplifier, also described in Chapter 5. Programs that have been verified with the prototype incremental verifier include a binary search program, which is first discussed in Chapter 3, and a selection sort. Finally, Chapter 6 summarizes this research and its results, and indicates some directions for further research.
CHAPTER 2

AN AXIOMATIC FRAMEWORK

Incremental verification requires that programs be able to express their own properties — both those that are expected to be true of the program (assertions) and those that are known to be true (lemmas). This can be accomplished by adding some special language constructs to the source language of programs to be verified. For instance, the keyword assert can be used to flag assertions, and the keyword lemma can be used to flag lemmas. During verification, the difference between assertions and lemmas is that assertions are to be proved true, but lemmas are assumed true. Thus, lemmas can be used to help prove assertions.

This chapter introduces some language constructs that can be used to incorporate assertions and lemmas into programs. Both informal and formal semantics are given for these constructs, an axiomatic approach (Hoare 1969; Hoare and Wirth 1973) being used to provide the formal semantic description. An axiomatic semantic description for programming language constructs is important for two reasons.

1. It provides a semantic definition for operations in the language, and
2. It provides a method for proving the correctness of programs written in the language.

Adding special language constructs to a programming language is not the only way of associating statements of program properties to programs, of course. For instance, an external data base could be used to store lemmas and/or assertions for programs. VISTA (German and Wegbreit 1975), for example, stores proven program properties in a data base. However, using special language constructs instead of some other method has an advantage: the roles that lemmas and assertions play during verification is made explicit by their roles in the semantic definition for the programming language. Consequently, when a formal semantic description drives the verification process, as is the case for axiomatic semantics, integrating assertions and lemmas into verification can be done quite smoothly, in a manner that is consistent with the rest of the programming language. In practice, incorporating lemmas and assertions into an axiomatic semantic definition for a programming language does not seem to be technically very difficult.

For those readers who are not entirely familiar with axiomatic semantic descriptions, this chapter begins with a discussion of axiomatic semantics.
Axiomatic Semantics

An axiomatic semantic description for a programming language is essentially a formal system of logic that provides a method for proving partial correctness for programs written in the language. Partial correctness for a program means that the program is correct whenever the program terminates. For convenience, such a formal logic is referred to below as a verification logic.

Underlying a verification logic are two other formal systems. One is the programming language whose semantics are of interest. The other is an assertion logic, which provides a way of talking about properties of programs written in the programming language. Usually, the assertion logic is an applied first-order predicate calculus.

Like any formal system of logic, a verification logic consists of:
1. A set of formulas,
2. A meaning for each formula, and
3. A proof theory.

Formula Syntax

The formulas in a verification logic include the formulas of the assertion logic, as well as verification statements. A verification statement is of the form:

\( \{P\} \text{ A } \{Q\} \)
where P (the **precondition**) and Q (the **postcondition**) are formulas in the assertion logic, and A is a fragment of a program written in the programming language. Not just any program fragment is permitted, however, so that the description of a particular verification logic must provide the appropriate restrictions.

**Formula Semantics**

The meaning of formulas in the verification logic that are also formulas in the assertion logic is as defined for the assertion logic. The meaning of \{P\} A \{Q\} is: if P is true immediately before executing A and if A terminates, then Q is true immediately after executing A. Thus, the meaning of a verification statement is derived from the meaning of its parts -- the meaning of the pre- and postconditions coming from the assertion logic and the meaning of the program fragment coming from the prescribed or actual behavior of programs in the programming language.

Formulas in the assertion logic are **about** properties of programs, with variables providing the link between formulas and programs. Thus, the set of variables available to the assertion logic must include the set of available program variables. Likewise, the ranges of values for variables in the assertion logic and programming logic must be compatible.
Proof Theory

In any formal system of logic, the proof theory provides a purely syntactic means for determining the truth of a formula. A proof is usually defined as a sequence of formulas such that each formula is either an axiom or is a direct consequence of previous formulas in the proof. A proof theory is \textbf{complete} if the set of true formulas is identical to the set of provable ones.

For a verification logic, the axioms consist of all the theorems, i.e., provable formulas, in the assertion logic and the (possibly empty) set of verification statements that have been designated as axioms. Rules of inference are of the form

\[
\begin{array}{c}
H_1, \ldots, H_n\\ \hline \\
H
\end{array}
\]

where \(H_1\) through \(H_n\) are formulas and \(H\) is a verification statement. A rule of inference states that \(H\) is a \textbf{direct consequence} of \(H_1\) through \(H_n\), i.e., if \(H_1\) through \(H_n\) are true, then so is \(H\).

A Clarification

From the above discussion, it would appear that a programming language provides, in part, meanings for verification statements in a verification logic. Since verification logics have been identified with axiomatic semantic
descriptions of programming languages, this conflicts with the conventional view in which an axiomatic semantic description provides meaning for the operations in a programming language.

The seeming conflict is easily resolved, however. When a formal logic is defined without an underlying model being specified to provide the formulas with meaning, the formal logic becomes a formal axiomatic theory. (See, for example, Kleene 1967.) Then one is free to choose any model that makes the axioms and rules of inference true. In this way, a verification logic becomes an axiomatic semantic description by leaving the semantics of the underlying programming language unspecified, except for the following constraint: any realization, i.e., implementation, of the language must make the axioms true and rules of inference valid. Equivalently, any implementation must be such that every provable formula is true for the implementation.

**Assertions**

An assertion is a formula, imbedded in the text of a program, that is expected to be true each time the assertion is encountered during program execution, i.e., each time control passes through it. An assertion is said to be valid if it actually is true at each encounter. Whenever a programming language provides for imbedding assertions in
programs, its verification logic usually is designed so that verification of the program implies validation of its assertions.

Input Conditions

Strictly speaking, in order for an assertion to be valid, it must be true regardless of the state in which program execution starts. In practice, however, programs are intended to be executed only when a certain input condition is satisfied, so that all proofs of correctness are done with respect to this input condition. Consequently, the notion of "valid assertion" can be extended: a valid assertion is an assertion that is true at each encounter in every execution of the program where the input condition is satisfied.

In order to permanently associate an input condition with a program, it is necessary that programs be able to express their own input conditions. For example, a programming language might require each program to begin with an entry statement of the form:

```
entry P;
```

where P is a formula in the assertion logic. Consider the following rule of inference for the entry statement:

```
{P} A {Q}  
------------
{R} entry P; A {Q}
```
where $P$, $Q$, and $R$ are formulas in the assertion logic and $A$ is the entire program excluding its entry statement. Provided that appropriate rules of inference are provided for imbedded assertions, the above rule ensures that the verification of a program validates the assertions of the program with respect to the input condition specified in the program's entry statement.

Imbedding Assertions in Programs

The original motivation for including assertions in the text of programs was to transfer the responsibility for discovering appropriate inductive assertions from an (automatic) verifier to the programmer. For instance, a rule of inference that suffices to verify programs containing while statements is:

$$
P \rightarrow I, \{I \& B\} S \{I\}, I \& B \rightarrow Q
$$

$$\frac{}{\{P\} \text{ while } B \text{ do } S \{Q\}}$$

where $P$, $I$ and $Q$ are formulas in the assertion logic, $B$ is a Boolean expression, and $S$ is some arbitrary program statement. This rule, however, is not quite suitable for practical use in automatic verification, since it leaves the choice of inductive assertion, i.e., $I$ in the above rule of inference, unspecified. That is, in order to prove a verification statement of the form

$$\{P\} \text{ while } B \text{ do } S \{Q\}$$
a formula \( I \) must be found for which the three formulas in
the top half of the rule can be proved true.

A solution to the problem is to extend the syntax of
the **while** statement to include an **assert** clause, which spec-
ifies an inductive assertion for the loop, e.g.

```
assert I while B do S
```

The semantics of this statement can be stated informally as
being the same as for the ordinary **while** statement, except
that \( I \) is expected to be true each time immediately before
executing the test \( B \). The following rule of inference for
the **assert-while** statement requires that an inductive asser-
tion that is sufficient to verify the **while** loop be speci-
fied.

\[
P \rightarrow I, \{I \& B\} S \{I\}, I \& \neg B \rightarrow Q
\]

\[
\text{------------------------}
\]

\[
\{P\} \text{ assert } I \text{ while } B \text{ do } S \{Q\}
\]

This rule can be applied entirely automatically.

Of special interest for incremental verification is
how the above rule of inference contributes to assertion
validation. The rule states that if the verification state-
ment is true, then

1. If \( P \) is true at entry to the loop, then \( I \) is also true
   at loop entry.

2. If \( I \) is true at the start of one execution of the loop
   body, then it is again true at the start of the next
   execution of the loop body.
In other words, assuming P is true immediately before execution of the loop, I is true at the start of each traversal of the loop. Thus, the process of proving the verification statement

\{P\} assert I while B do S \{Q\}

validates the inductive assertion I, with respect to the precondition P.

Moreover, I is valid for the program in which the while statement is imbedded, with respect to the input condition specified in the program's entry statement. This claim is justified as long as the rules of inference act in concert so that P is in some sense simply a propagation of the program's input condition through the program text to the while statement.

The idea of validating programmer-specified assertions in the course of verifying a program can be extended to non-inductive assertions as well. For instance, a programming language might include an assert statement of the form:

assert R

where R is a formula in the assertion logic. Informally, the semantics of this statement is simply that R is expected to be true each time the statement is encountered during program execution (assuming that the program is only executed when its input condition is satisfied). The following
rule of inference provides a formal expression of the semantics of the assert statement:

\[ P \rightarrow R \& Q \]

\[ \{ P \} \text{ assert } R \{ Q \} \]

Notice that the rule has been designed so that proof of the verification statement in the bottom half of the rule implies validation of the assertion R with respect to the precondition P. Moreover, the assertion is valid for the program in which the assert statement is imbedded, with respect to the program's input condition.

Lemmas

A lemma is an assertion that is known to be valid. Suppose a programming language provides a syntactic construct for specifying lemmas, distinguishing them from unvalidated assertions. Then in order for a program to be well-formed, any lemmas appearing in it must have been validated somehow. For such a programming language, a verification logic can be designed so that lemmas are used to advantage in the validation of other assertions.

For example, a lemma-assert statement might be included in the programming language, having the form:

\[ \text{lemma } P \text{ assert } Q \]

where P and Q are formulas in the assertion logic. Informally, the statement's semantics can be stated as follows:
P is known to be valid with respect to the input condition specified in the program's entry statement, and Q is expected to be valid with respect to the same input condition. The following rule of inference gives semantics for the lemma-assert statement more formally:

\[ R_1 \land P \rightarrow Q \land R_2 \]

\{R_1\} lemma P assert Q \{R_2\}

This rule corresponds to our intuitive understanding of the lemma-assert statement. Since the lemma clause is not executable and since P is known to be true whenever the statement is encountered, the verification statement in the bottom half of the rule is true if the following is true:

\{R_1 \land P\} assert Q \{R\}

By the rule of inference given earlier for the assert statement, this reduces to proving \(R_1 \land P \rightarrow Q \land R_2\), which is the formula appearing in the top half of the rule of inference for the lemma-assert statement.

The effect of this rule is to take advantage of the valid assertion P in order to validate the assertion Q. Note that if the lemma clause did not appear, then the verification condition required to validate Q would be \(R_1 \rightarrow Q \land R_2\). Even if P is implied by R1, this verification condition may be more difficult to prove than \(R_1 \land P \rightarrow Q \land R_2\), since in the latter formula P appears explicitly as an antecedent to the implication.
Valid inductive assertions might be incorporated into a program text with a while statement that has been augmented with a lemma clause in addition to an assert clause. The syntax for such a statement might be:

```
lemma P assert Q while B do S
```

where P and Q are formulas in the assertion logic, B is a Boolean expression and S is an arbitrary program statement. Informally, the semantics of the lemma-assert-while statement are exactly the semantics of the assert-while statement as discussed earlier, except that P is known to be true each time immediately before executing the test B.

The following rule of inference gives a formal semantic description of the lemma-assert-while statement (referred to henceforth simply as the while statement):

```
R1->Q, {P&Q&B} S {Q}, P&Q&~B -> R2
```

This rule corresponds to our intuitive understanding of the statement. Note that since the lemma clause is not executable, the verification statement in the bottom half of the rule is true if the following is true:

```
{R1} assert Q while B do S {R2}
```

By the semantics given earlier for the assert-while statement, this reduces to proving three other formulas:

```
R1->Q
{Q&B} S {Q}
Q&~B -> R2
```
However, since P is known to be true immediately before executing the test B, the last two formulas above can be replaced by:

\[
\{P \land Q \land B\} \Rightarrow \{Q\}
\]

\[
P \land Q \land \neg B \Rightarrow R_2
\]

Thus, proving the three formulas in the top half of the rule of inference for the \textit{while} statement is sufficient to prove the verification statement in the bottom half.

Like the rule of inference for the \textit{lemma-assert} statement, the rule of inference for the \textit{while} statement takes advantage of the known validity of the formula in the \textit{lemma} clause to validate the formula in the \textit{assert} clause. The valid assertion is used to strengthen
1. The precondition of the verification statement that expresses the inductiveness of the inductive assertion, i.e.,

\[
\{Q \land B\} \Rightarrow \{Q\} \text{ becomes } \{P \land Q \land B\} \Rightarrow \{Q\}
\]

2. The antecedent of the formula that expresses the validity of the postcondition, i.e.,

\[
Q \land \neg B \Rightarrow R_2 \text{ becomes } P \land Q \land \neg B \Rightarrow R_2
\]

In the remainder of this dissertation, a programming language referred to as L is used to illustrate incremental verification and assertion synthesis. Informally, it suffices to describe L as an Algol-like language whose domain
is the integers and singly-dimensioned arrays of integers. The \texttt{lemma-assert} and \texttt{lemma-assert-while} statements introduced above are statements in L. L does not contain \texttt{assert} and \texttt{assert-while} statements as such, but considers these to be special cases of the two \texttt{lemma} statements; whenever a \texttt{lemma} or \texttt{assert} clause is omitted, the missing formula defaults to \texttt{TRUE}. A formal description of the syntax and semantics of L appears in Appendix A, and a description of the assertion logic for L appears in Appendix B.
CHAPTER 3

DEVELOPING INDUCTIVE ASSERTIONS

The inductive assertions required to verify even a short, simple program often are surprisingly complex. The purpose of this chapter is to point out some of the sources of complexity and to show how an incremental approach to assertion synthesis can be used to develop inductive assertions in spite of this complexity. That the complexity of inductive assertions can be dealt with in an effective way is suggested by the following empirical observation: most inductive assertions are merely conjunctions of individual loop invariants. It is shown below that these loop invariants play a limited number of roles and that it is possible to develop inductive assertions by conjoining loop invariants one by one as the need for them becomes evident.

Terminology

Consider a while statement W having the following form:

\[ \text{while } B \text{ do } S \]

Any formula K that satisfies \([K&B]\text{ while } B \text{ do } S \{K\}]\) is said to be inductive over W.
An **inductive invariant** of a loop \( W \) is a formula that is true at entry to \( W \) and is inductive over \( W \). Thus, an inductive invariant is true after each traversal of the loop.

An **invariant** of \( W \) is any formula \( W \) that is implied by an inductive invariant of \( W \). Thus, an invariant is true at entry to the loop and after each traversal, but might not by itself be inductive over the loop.

In a `while` statement of the form:

```plaintext
lemma I assert J
while B do S
```

I is the **verified invariant** of the loop, and \( J \) is the **asserted invariant**. The formula formed by conjoining \( J \) to \( I \), i.e., \( I \& J \), is referred to as the **loop assertion**. A loop assertion corresponds to the usual notion of "inductive assertion" for `while` loops, except that a loop assertion has a component that has previously been verified as being invariant.

**Exit Assertions**

If a partially developed loop assertion is not strong enough to prove the exit condition for a loop, then an additional loop invariant must be found that can be conjoined to the loop assertion to prove the exit condition.

For instance, consider the following program, which computes \( n! \). In the program, terms of the form `fact(z)` reference the factorial function. (Since semantics for `fact` are not built
into L, it is assumed that an interpretation for fact will be made available when the program is verified.) The programmer has already provided what seems to be a reasonable loop assertion.

```plaintext
entry 0<n;
ex:=1; i:=0;
assert x=fact(i)
while i<n do
  begin i:=i+1; x:=x*i end;
assert x=fact(n)
```

Although the loop assertion is an inductive invariant, it is not strong enough to prove the exit assertion, i.e., $x=fact(n)$ cannot be proved from $x=fact(i) \land \neg (i<n)$. The problem can be solved by conjoining $i\leq n$, which is another inductive invariant for the loop, to the loop assertion.

Just as the invariant $x=fact(i)$ is closely related to the "primary effect" of the loop, which is to assign the value $fact(n)$ to $x$, so the invariant $i\leq n$ is closely related to a "secondary effect", which is to set the number of traversals of the loop at $n$. Notice that the primary effect is dependent upon the secondary effect; that is, the fact that the loop assigns to $x$ the value $fact(n)$ depends on the fact that the loop is executed exactly $n$ times.

In general, separable effects of a loop contribute individual invariants to its inductive assertion. When the complexity of an inductive assertion arises partially due to secondary effects, it may be beneficial to use incremental
assertion synthesis to first verify the loop with respect to the secondary effect, and in a later pass verify the loop with respect to the primary effect. In this manner, the invariants may be conjoined to the loop assertion one at a time in separate passes by stating each effect of the loop as a separate loop exit assertion.

This approach can be illustrated with the factorial program. For the first pass, the programmer has not provided any loop assertion, letting it default to TRUE, and has specified the loop's secondary effect as the exit assertion rather than the primary effect.

```plaintext
entry 0<n;
  x:=1; i:=0;
  while i<n do
    begin i:=i+1; x:=x*i end;
  assert i=n
```

Since TRUE is too weak an invariant to verify this loop with respect to the exit assertion i=n, an invariant that is strong enough to prove the exit assertion needs to be found. The required invariant is i\leq n. Conjoining this invariant to the existing loop assertion gives TRUE \& i\leq n, which simplifies to i\leq n.

The next pass develops an inductive assertion that is strong enough to verify the primary effect of the loop, i.e., x=\text{fact}(n). The input program is the output program from the previous pass, augmented with a new exit assertion.
entry 0≤n;
x:=1; i:=0;
lemma i≤n
while i<n do
  begin i:=i+1; x:=x*i end;
lemma i=n assert x=fact(n)

Now the secondary effect of the loop can be used to aid in finding an invariant that can be conjoined to the existing loop assertion to verify the exit assertion. Since i=n is already known to be true when the loop terminates, the loop exit assertion x=fact(n) would be verified if x=fact(i) were an invariant of the loop. Since x=fact(i) is in fact an invariant, conjoining x=fact(i) to the existing loop assertion yields an inductive assertion that is sufficient to verify the program.

Note that since i=n is known to be true when the loop terminates, in order to prove the exit assertion it suffices to prove the weaker exit assertion

i=n → x=fact(n)

Thus, even if the existing loop assertion were simply TRUE, conjoining x=fact(i) to it would still be sufficient to verify the program. Thus, a loop's exit lemma may make it possible to verify the loop's exit assertion using a much simpler inductive assertion than would be necessary if the exit lemma were not available.

This example shows how lemmas provide useful information for constructing trial loop assertions. In
particular, if a lemma appears immediately after a loop, the following heuristic may be applied:

**exit heuristic:** If an inductive assertion is needed to verify a loop with exit assertion Q and if P is known to be true when the loop terminates, find an R such that P&R→Q. The loop is verified if R is an invariant.

In the example, the exit heuristic is applied with x=fact(n) substituted for Q and i=n substituted for P, suggesting x=fact(i) as the needed invariant.

---

**Non-Inductive Invariants**

When using a heuristic approach to synthesizing assertions, it is common to encounter invariants that are not inductive. A non-inductive invariant can always be made inductive by conjoining an appropriate formula which is itself an invariant. That such a formula exists follows directly from the definition of invariant. Let I be a non-inductive invariant of a while loop

```
while B do S
```

Then there is an inductive invariant K for the loop such that K→I. If K is conjoined to I to obtain I'=K&I, then I' is clearly inductive.

In practice, however, it is often possible to find a formula that does not subsume I, but when it is conjoined to I nevertheless makes I inductive. For instance, consider a program:
entry P;
assert I
while B do S;
assert Q

Suppose I is true at entry to the loop, i.e., P\rightarrow I, I is strong enough to verify the exit assertion, i.e., I\&\neg B \rightarrow Q, but, although I appears to be a plausible invariant, it is not inductive. If an invariant R for the loop can be found such that \{R\&I&B\} S \{I\} can be proved, then conjoining R to I makes I inductive.

When the incremental verifier encounters a candidate invariant I that is not inductive, the inductive assertion can be developed piecemeal in the following way. If a candidate invariant J can be found such that \{J\&I&B\} S \{I\}, the incremental verifier calls itself recursively to verify that J is an invariant. Before calling itself recursively, the verifier temporarily erases from the program all programmer-specified assertions that have been seen so far and inserts an assertion for J. For the program above, the input program to the recursive call to the verifier is:

entry P;
assert J
while B do S

If the recursion succeeds in verifying that J is an invariant, then on return the while loop has been augmented by the clause lemma J, effectively making I inductive. The verifier replaces all those assertions that were removed before the recursion, and continues the verification.
An example of a program in which non-inductive invariants arise quite naturally is the following one, which performs a binary search:

```
entry b<c & issorted(a[b:c]);
while b<>c do
begin
  d:=(b+c)/2;
  if key>a[d] then b:=d+1 else c:=d
end
```

The construct a[b:c] is the (sub)array a[b], ..., a[c], provided that c is not less than b. The predicate issorted is true when its array argument is sorted into ascending order.

In order to express the correctness of the program, it is necessary to "freeze" the initial values of b and c. If the statements

```
let b0:=b; let c0:=c;
```

are inserted immediately after the `entry` statement, then an exit assertion that expresses a necessary condition for the program to be correct is:

```
isin(key,a[b0:c0]) -> key=a[b]
```

The predicate `isin` is true if its first argument occurs as an element in the array specified by its second argument.

The loop works by adjusting the values of b and c so that, if key occurs in a[b0:c0], then immediately before executing the body of the loop, key occurs in a[b:c]. This suggests that an invariant of the loop might be:

```
isin(key,a[b0:c0]) ->isin(key,a[b:c])
```
Suppose the following program text is passed to the incremental verifier:

```plaintext
entry b< c & issorted(a[b:c]);
let b0:=b; let c0:=c;
assert isin(key,a[b0:c0]) -> isin(key,a[b:c])
while b≠c do
begin
    d:=(b+c)/2;
    if key>a[d] then b:=d+1 else c:=d
end;
assert isin(key,a[b0:c0]) -> key=a[b]
```

While processing this program, the incremental verifier determines that the asserted invariant, which may be abbreviated as I, is strong enough to prove the exit assertion when it proves the following verification condition:

\[
I \land \neg(b\neq c) \rightarrow (isin(key,a[b0:c0]) \rightarrow key=a[b])
\]

i.e.,

\[
(isin(key,a[b0:c0]) \rightarrow isin(key,a[b:c])) \land b=c \\
\rightarrow (isin(key,a[b0:c0]) \rightarrow key=a[b])
\]

However, the asserted invariant is found not to be inductive as a result of trying to verify

\[
\{I \land b\neq c\} \\
d:=(b+c)/2; \\
if key>a[d] then b:=d+1 else c:=d \\
\{I\}
\]

which reduces to the following verification condition:

\[
(isin(key,a[b0:c0]) \rightarrow isin(key,a[b:c])) \land b\neq c \\
\rightarrow \ (if key>a[(b+c)/2]
\quad then (isin(key,a[b0:c0]) \rightarrow isin(key,a[((b+c)/2)+1:c]))
\quad else (isin(key,a[b0:c0]) \rightarrow isin(key,a[b:(b+c)/2]))
\]

The above formula simplifies to:
\[ z = \frac{(b+c)}{2} \]
\[
\Rightarrow \quad b \neq c \land \text{isin}(key, a[b:0:c]) \land \text{isin}(key, a[b:c])
\]
\[
\Rightarrow \quad \begin{cases} 
\text{if } \text{key} > a[z] \\
\text{then } \text{isin}(key, a[z+1:c]) \\
\text{else } \text{isin}(key, a[b:z])
\end{cases}
\]

Since this formula is not provable, the asserted invariant is not inductive. However, the formula is provable under the hypothesis \( H \), where \( H \) is the formula

\[ \text{issorted}(a[b:c]) \land b < c \]

Thus, if \( H \) is an invariant of the loop, then the asserted invariant can be made inductive by conjoining \( H \) to it.

Assume the verifier is able to generate \( H \) as the needed invariant, perhaps by simply asking the user for suggestions. Then the verifier recursively invokes itself to verify the following program:

```
entry b < c & issorted(a[b:c]);
let b0 := b; let c0 := c;
assert issorted(a[b:c]) & b < c
while b \neq c do
begin
\( d := \frac{(b+c)}{2} \);
\quad \text{if } \text{key} > a[d] \text{ then } b := d + 1 \text{ else } c := d
end
```

Without going into further detail, this pass of the verifier establishes that \( H \) is inductive and invariant. When control finally returns to the original verification, the \textbf{while} statement has been augmented with the clause

\[ \text{lemma } \text{issorted}(a[b:c]) \land b < c \]

Since the asserted invariant
isin(key, a\[b_0:c_0\]) \rightarrow isin(key, a\[b:c\])

can be proved true at entry to the loop, an inductive assertion that is sufficient to verify the program has been developed. The output of the incremental verifier is:

entry b < c & issorted(a\[b:c\]);
let b_0 := b; let c_0 := c;
lemma issorted(a\[b:c\]) & b < c
& isin(key, a\[b_0:c_0\]) \rightarrow isin(key, a\[b:c\])
while b ≠ c do
begin
\(d := \frac{(b+c)}{2};\)
if key > a\[d\] then \(b := d + 1\) else \(c := d\)
end;
lemma isin(key, a\[b_0:c_0\]) \rightarrow key = a\[b\]

**Nested Loops**

The requirement that a loop assertion be inductive and strong enough to verify the loop exit condition forces much interaction among inductive assertions for nested loops. For example, consider the following program:

entry 0 < n;
\(x := 1; i := 0;\)
while i < n do
begin
\(j := x;\)
while 0 < j do
begin
\(x := x + 1;\)
\(j := j - 1\)
end;
\(i := i + 1\)
end

The outer loop computes \(2^n\) by iteratively doubling the value of \(x\), the doubling being done by the inner loop. To verify
that \( x=2^n \) when the outer loop terminates, inductive assertions for both loops are required. The following is a step-by-step account of how the necessary inductive assertions might be developed without using an incremental verifier.

1. Upon entry to the outer loop, \( x=2^0 \). After one traversal of the loop, \( x=2^1 \), after two traversals, \( x=2^2 \), and so on. Since \( i \) counts the number of traversals, \( x=2^i \) is a good approximation to the required inductive assertion for the outer loop.

2. The assertion \( x=2^i \), however, is not strong enough to prove \( x=2^n \) at termination, i.e., \( x=2^n \) cannot be proved from \( -(i<n) \& x=2^i \). Conjoining \( i\leq n \), which is easily seen to be an inductive invariant for the outer loop, to the loop assertion solves the problem. Thus, the loop assertion tentatively developed so far for the outer loop is

\[
x=2^i \& i\leq n
\]

3. In order to verify that \( x=2^i \& i\leq n \) is inductive over the outer loop, it is necessary to verify

\[
\{x=2^i \& i\leq n \& i<n\}
\]

\[
j:=x;
\]

\[
\text{while } 0<j \text{ do}
\]

\[
\begin{align*}
\text{begin} \\
x&:=x+1; \\
j&:=j-1 \\
\text{end}; \\
i&:=i+1
\end{align*}
\]

\[
\{x=2^i \& i\leq n\}
which poses the problem of finding a satisfactory inductive assertion for the inner loop. This inductive assertion must be strong enough to prove that, when the loop terminates, \( x=2^{i+1} \& i+1 \leq n \) is true. Each conjunct of the loop exit condition may be treated separately.

4. The inductive assertion needed for the inner loop to verify that \( i+1 \leq n \) is true at loop exit is \( i+1 \leq n \) itself. Since neither \( i \) nor \( n \) changes its value during execution of the loop, \( i+1 \leq n \) is inductive over the loop. Moreover, \( i+1 \leq n \) is true whenever the loop is entered.

5. Finding an invariant for the inner loop to verify \( x=2^{i+1} \) at loop exit is a bit more difficult. Note that the loop increments \( x \) as it decrements \( j \), so that the sum \( x+j \) remains constant throughout the loop. Since \( x+j \) equals \( 2^{i+1} \) at entry to the loop, a good guess for the required invariant appears to be \( x+j=2^{i+1} \). Conjoining the two invariants computed so far for the inner loop yields the following tentative loop assertion:

\[
x+j=2^{i+1} \& i+1 \leq n
\]

6. Unfortunately, this loop assertion is not quite strong enough to verify that \( x=2^{i+1} \) is true when the inner loop terminates. The problem is easily solved by conjoining \( 0 \leq j \) to the loop assertion. \( 0 \leq j \) is strong enough to
verify the exit condition $x=2^i+1$ and is inductive over the loop.

7. However, in order to verify that $0<j$ is true at entrance to the inner loop, the loop assertion for the outer loop needs to be strengthened. Conjoining $0<x$ to the outer loop's loop assertion will suffice if $0<x$ can be shown to be an invariant of the outer loop.

8. $0<x$ is true at entry to the outer loop. Verifying that it is inductive necessitates that the loop assertion for the inner loop be strengthened. For this purpose, $0<x$, which is another invariant of the inner loop, can be conjoined to the loop assertion for the inner loop.

The inductive assertions that have been developed for the nested loop program are

$$x=2^i \ & \ i<n \ & \ 0<x$$

for the outer loop and

$$i+1<n \ & \ x+j=2^i+1 \ & \ 0<j \ & \ 0<x$$

for the inner loop.

This simple program with only two nested loops demonstrates that developing inductive assertions becomes very complicated due to interactions between the assertions. In general, whenever an (inner) loop is enclosed in another (outer) loop, the inner loop's inductive assertion must be strong enough to verify that the loop assertion of the outer
loop is inductive. Often an invariant is conjoined to the loop assertion of the inner loop for every invariant conjunct of the outer loop's assertion. (See steps 4, 5, 6, and 8 of the example.) Additionally, whenever an (outer) loop encloses another (inner) loop, the outer loop's inductive assertion must be strong enough to verify that the inner loop's inductive assertion is true at entry. This may necessitate that additional invariants be conjoined to the loop assertion for the outer loop. (See step 7 of the example.)

Even though the synthesis of inductive assertions for nested loops is quite complex, the assertions can be developed in a fairly straightforward manner by conjoining invariants to partially developed loop assertions as they are needed. As discussed earlier, an incremental approach to assertion synthesis simplifies the process if the user takes care to separate primary effects from secondary ones and if recursion is used to augment non-inductive assertions with the required invariants. A program with multiple loops provides another opportunity to take advantage of the incremental approach: since a program is correct only if each of its loops is correct, an obvious application of the incremental approach is to verify separately that each loop does what it is expected to do. If inner loops are processed before outer ones, then the inductive assertions for all loops are built up step-by-step as more loops are verified.
For example, incremental assertion synthesis can be applied to the nested loop example program so that in each pass inductive assertions are synthesized gradually. The program is best attacked by verifying the inner loop before the outer one, and secondary effects before primary effects.

The secondary effect of the inner loop is to decrement \( j \) until, when the loop terminates, \( j=0 \). Thus, the first pass of the incremental verifier proves that \( j=0 \) at the end of the inner loop, constructing the required inductive assertions. Verifying the inner loop's exit assertion necessitates that a sufficiently strong loop assertion be added to the loop. The required assertion is easily seen to be \( 0 < j \), which is inductive over the inner loop and is strong enough to prove the loop exit assertion, \( j=0 \). However, if \( 0 < j \) is the loop assertion of the inner loop, then the outer loop's loop assertion, which at this point is the default assertion \( \text{TRUE} \), is not inductive. For, in order for \( \text{TRUE} \) to be inductive, the following must be provable:

\[
\{ \text{TRUE} \land i < n \}
\]
\[
j := x;
\]
\[
\text{assert } 0 < j
\]
\[
\text{while } 0 < j \land i < n \text{ do}
\]
\[
\begin{align*}
\text{begin} & \\
x & := x + 1; \\
j & := j - 1 \\
i & := i + 1
\end{align*}
\]
\[
\{ \text{TRUE} \}
\]

This reduces to verifying

\[
\{ \text{TRUE} \land i < n \} \quad j := x \quad 0 < j
\]
which reduces to the verification condition

\[ \text{TRUE } \& \ i < n \rightarrow 0 \leq x \]

Although this verification condition cannot be proved, it can be proved under the hypothesis \(0 < x\), which is another plausible invariant of the outer loop. Thus, the incremental verifier recursively calls itself to verify \(0 < x\) as an invariant, and it succeeds. Consequently, the original verification succeeds and outputs the following program text:

```
entry 0 < n;
x := 1; i := 0;
lemma 0 < x
while i < n do
begin
  j := x;
  lemma 0 < x \& 0 < j
  while 0 < j do
  begin
    x := x + 1;
    j := j - 1
  end;
  lemma j = 0;
i := i + 1
end
```

The next pass verifies that the primary effect of the inner loop is to double the value of \(x\). For this purpose the programmer inserts the statement

\`
let x0 := x;
```

immediately before the inner loop and adds the clause

\`
assert x = x0 + x0
```

to the \texttt{lemma} statement immediately following the loop.

Since the loop assertion developed so far for the inner loop
is not strong enough to verify the new loop exit assertion, a satisfactory invariant for the inner loop must be found. The loop increments \( x \) as it decrements \( j \), so that the sum of \( x \) and \( j \) remains constant. Using this information and the fact that \( j=0 \) when the loop terminates, the exit heuristic can be applied to yield \( x+j=x_0+x_0 \) as a possible loop invariant, which it is. This time, no additional invariant needs to be added to the loop assertion of the outer loop. This pass outputs the following program text:

```plaintext
entry 0<n;
ex:=1; i:=0;
lemma 0<x
while i<n do
begin
  j:=x;
  let x0:=x;
  lemma 0<x & 0<j & x+j=x0+x0
  while 0<j do
  begin
    x:=x+1;
    j:=j-1
  end;
  lemma j=0 & x=x0+x0;
i:=i+1
end
```

The next pass verifies the secondary effect of the outer loop, i.e., that when the loop terminates, \( i=n \). The effect of this pass conjoins \( i\leq n \) to the outer loop assertion, \( i+1\leq n \) to the inner loop assertion, and adds the statement

```
lemma i=n
```

to the end of the program.
The final pass verifies that at the end of the outer loop, \( x = 2^n \). The exit heuristic suggests \( x = 2^i \) as a satisfactory loop invariant for the outer loop. Verifying that \( x = 2^i \) is inductive presents the problem of finding an invariant for the inner loop that is sufficient to prove \( x = 2^{i+1} \) when the inner loop terminates. This time the exit heuristic can be applied to yield \( x_0 = 2^i \) as the needed loop invariant, since \( x = 2^{i+1} \) can be proved from \( j = 0 \) & \( x = x_0 + x_0 \) if \( x_0 = 2^i \). The proposed invariant is indeed invariant since \( x_0 \) is unchanging in the inner loop, and the loop assertion on the outer loop remains inductive. Thus, the pass is successful, and the incremental verifier outputs the following:

```plaintext
entry 0<n;
x:=1; i:=0;
lemma 0<x & i<n & x=2^i
while i<n do
begin
j:=x;
let x0:=x;
lemma 0<x & 0<j & x+j=x0+x0 & i+1<n & x0=2^i
while 0<j do
begin
x:=x+1;
j:=j-1
end;
lemma j=0 & x=x0+x0;
i:=i+1
end;
lemma i=n & x=2^n
```

Separately verifying that each program loop does what it is supposed to has several advantages.

1. The statement of a loop's effect may simplify finding a
needed invariant for the loop by providing information that is difficult to extract otherwise. Verifying loops individually permits the user to provide a simple statement of the loop's expected effect, which can then be used by heuristic methods for generating trial loop invariants.

2. The exit heuristic becomes available as another heuristic to be used in generating trial loop invariants for inner as well as outer loops.

3. The inductive assertions for all loops are built up gradually in a fairly transparent way.

4. If a program loop is incorrect, this fact becomes evident in the course of attempting to verify the loop, since the verification eventually fails when adequate inductive assertions cannot be found. Verifying loops individually serves to isolate incorrect code.

It was noted earlier that neither the analytic methods for synthesizing inductive assertions nor the heuristic methods perform well for programs containing nested loops. Using an incremental approach with heuristic methods, however, provides a promising method for dealing with nested loops. Whereas the inside-out assertion synthesis methods lose information in the course of progressing
from inner loops to outer loops, incremental assertion synthesis actually gains information as it progresses from inner loops to outer ones, if separate passes are used to verify the loops. Ordinarily, heuristic methods rapidly lose control of the interdependencies that develop between inductive assertions of nested loops; performing the synthesis incrementally constructs the interdependent inductive assertions in a controlled manner that is partly directed by the user.
A verification system that uses heuristic methods for synthesizing assertions must contain the following two components: a generator of trial inductive assertions and a controller. The controller part decides what part of the program being verified should be processed next, determines whether a trial assertion is valid, requests alternate trial assertions, etc. The incremental verifier presented in this dissertation can be viewed as a controller for heuristic assertion synthesis methods.

This chapter describes the behavior of the incremental verifier by presenting the verifier's underlying algorithm. This algorithm processes the program in a top-down fashion, proving verification conditions one-by-one as they are generated. Whenever a verification condition cannot be proved, it is because some inductive assertion is inadequate. When this happens, the algorithm backs up, generates an alternate inductive assertion by appealing to heuristic methods for assertion synthesis, and continues from the backup point. The basis of the algorithm is the computation of weakest preconditions.
Weakest Preconditions

Let A be a fragment of a program written in a given programming language that has an associated axiomatic semantic description, and let Q be a formula in the assertion logic. Then the weakest precondition of A given Q, written [A]Q, is a formula P such that

1. P satisfies {P}A{Q}, and
2. For all P' satisfying {P'}A{Q}, P' -> P.

Note that this definition of "weakest precondition" differs from Dijkstra's (1975), which guarantees termination of the program fragment. Also, the notation [A]Q should not be confused with V. Pratt's notation for formulas in dynamic logic (Pratt 1976), which is a modal logic for programs.

The following theorem suggests a method for verifying programs, provided there is an effective way of computing the weakest precondition: in order to verify {P}A{Q}, first compute [A]Q and then prove P -> [A]Q.

Theorem 1. {P}A{Q} if and only if P -> [A]Q.

Proof. (Only if) Assume {P}A{Q}. By definition, [A]Q is the weakest of all preconditions of A given the postcondition Q. Thus, since P is a precondition of A given the postcondition Q, [A]Q is as weak or weaker than P, i.e., P -> [A]Q.

(If) Assume P -> [A]Q. The following rule of inference may be assumed to hold for any satisfactory axiomatic definition of a programming language (Floyd 1967):
\( P \rightarrow R, \{R\} A \{Q\} \)

\[ \frac{}{\{P\} A \{Q\}} \]

Since \( \{[A]Q\}A\{Q\} \) is true by definition, the rule allows us to conclude \( \{P\}A\{Q\} \). \( \square \)

Appendix A presents an algorithm for computing weakest preconditions for \( L \). The function definition is recursive and by cases, one case for each rule of inference in the axiomatic semantics for \( L \). In fact, each case of the function definition is derivable in a fairly straightforward manner (with a little intuition) from the rules of inference. For instance, consider the rule of inference for a list of program statements:

\[ \{P\} A \{R\}, \{R\} S \{Q\} \]

\[ \frac{}{\{P\} A; S \{Q\}} \]

The rule suggests that \( [A;S]Q \) can be computed by a divide-and-conquer approach: first compute a formula \( R \) equal to \( [S]Q \), and then compute \( [A]R \), which is the desired weakest precondition. Another example comes from the rule of inference for simple variable assignment:

\[ P \rightarrow Q(x/fx) \]

\[ \frac{}{\{P\} x:=fx \{Q\}} \]

where \( Q(x/fx) \) is the formula obtained from \( Q \) by replacing all free occurrences of \( x \) by \( fx \). Clearly, since the formula \( Q(x/fx) \rightarrow Q(x/fx) \) is true, it is the case that \( Q(x/fx) \) is a precondition for \( x:=fx \) with postcondition \( Q \); intuition
suggests that it is also the weakest precondition. Appendix C provides a proof that the weakest precondition function as defined for L actually computes the weakest precondition.

The Algorithm

The algorithm underlying the incremental verifier is based on the weakest precondition function for L given in Appendix A. As noted earlier, the algorithm is a backtracking one. In order to clearly express its backtracking behavior, the algorithm is presented below with the help of an analogy. This analogy was suggested by a similar analogy used by Floyd to describe a backtracking parser (Floyd 1964).

Suppose a person is assigned the goal of validating the assertions in a program P written in L. This person has the power to hire subordinates, assign them tasks, and fire them if they fail; they in turn have the same power.

By convention, the subordinates are each hired to find the weakest precondition of a certain portion of the program P, for a given postcondition Q. After they have been appointed, the subordinates are asked to deliver their weakest preconditions to their superiors. If their weakest preconditions are found to be unsatisfactory, the subordinates may repeatedly be asked for alternate weakest preconditions. On the other hand, if their preconditions are
found to be satisfactory, the subordinates are expected to change the assertions in their individual program segments to lemmas.

In order for the subordinates to satisfy their superiors when faced with the seemingly unreasonable demand for an alternate weakest precondition, the subordinates are permitted to replace any asserted invariants appearing in their individual segments of program text by other formulas. These other formulas are generated by heuristic methods for assertion synthesis.

In this algorithm, each person corresponds more or less to a case in the definition of the weakest precondition function. Thus, the chief executive, i.e., the person assigned the original goal of validating the assertions in $P$, corresponds to the definition for $[\text{entry } R; A]Q$. The subordinates correspond to the remaining cases of the definition, the particular case depending on the form of the program segment assigned to the subordinate. The more interesting cases are detailed below, with some explanation. All the cases are presented in bulk, without explanatory material, in Appendix D.

The Chief Executive

Since $P$ is a program in $L$, $P$ has the form $\text{entry } R; A$
where $R$ is a formula in the assertion logic and $A$ is a list of statements in $L$. In order to validate the assertions in $P$, it suffices to verify $P$ with respect to the weakest possible postcondition, i.e., TRUE. Appealing to the weakest precondition function for $L$, this amounts to proving $R \rightarrow [A] \text{TRUE}$. Thus, the chief executive hires a subordinate $\text{SUB}_A$, whose task is to find weakest preconditions for $A$ given the postcondition TRUE. If $\text{SUB}_A$ delivers a satisfactory precondition, i.e., one that is implied by $R$, then $\text{SUB}_A$ is entitled to change any imbedded assertions in $A$ to lemmas. The chief executive behaves according to the following algorithm:

BEGIN
  hire a subordinate $\text{SUB}_A$ to find $[A] \text{TRUE}$;
  FIRST precondition $Q'$ received from $\text{SUB}_A$
    SUCHTHAT $R \rightarrow Q'$ is provable
    DO BEGIN
      tell $\text{SUB}_A$ that $Q'$ is satisfactory;
      succeed
    END
    OTHERWISE fail
END

The flow diagrams corresponding to the control structures used in this and the other algorithms presented in this dissertation can be found in Figure 1. Credit for the FIRST-SUCHTHAT control structure belongs to Alphard (Wulf et al. 1976).

Subordinates

A lemma-assert statement. Let $\text{SUB}_{\text{lemma}}$ be a subordinate who is to find weakest preconditions for a statement
Figure 1. Flow Diagrams for Control Structures
of the form:

```
lemma R1 assert R2
```

given the postcondition Q. Since a lemma-assert statement contains no asserted invariants, which could be changed to yield alternate weakest preconditions, the only possible weakest precondition is $R_1 \rightarrow R_2 \& Q$ (see the definition of the weakest precondition function for L). If SUBlemma's superior finds $R_1 \rightarrow R_2 \& Q$ to be satisfactory, then the assertion $R_2$ must be converted to a lemma. This is done by removing $R_2$ from the assert clause and conjoining $R_2$ to $R_1$ in the lemma clause. More precisely:

```
BEGIN
let Q' be $R_1 \rightarrow R_2 \& Q$;
succeed, returning Q';
IF my superior finds Q' satisfactory
THEN replace my program segment by "lemma R1&R2"
ELSE fail
END
```

A list of statements. Consider a subordinate

SUBlist, who is to find weakest preconditions for a list of statements given Q as the postcondition. That is, SUBlist is to find $[A;S]Q$, where A is a list of statements and S is a statement. Recall that according to the definition of the weakest precondition function for L, $[A;S]Q$ is equal to $[A][S]Q$. Thus, the strategy used by SUBlist is to hire two subordinates, SUB_A and SUB_S. SUB_S returns weakest preconditions for S given the postcondition Q, and SUB_A returns weakest preconditions for A given one of the preconditions
returned by $\text{SUB}_g$ as the postcondition. Each time $\text{SUB}_\text{list}$ is asked to deliver another weakest precondition, it returns one of the preconditions returned by $\text{SUB}_A$. If $\text{SUB}_\text{list}$'s superior is satisfied with a returned precondition, both $\text{SUB}_A$ and $\text{SUB}_g$ must be told that their last delivered preconditions are satisfactory, so that the subordinates can convert any assertions imbedded in their program segments to lemmas.

BEGIN
  hire a subordinate $\text{SUB}_g$ to find $[S]Q$;
  WHILE $\text{SUB}_g$ returns a precondition $Q'$ DO
    BEGIN
      hire a subordinate $\text{SUB}_A$ to find $[A]Q'$;
      WHILE $\text{SUB}_A$ returns a precondition $Q''$ DO
        BEGIN
          succeed, returning $Q''$;
          IF my superior finds $Q''$ satisfactory
            THEN BEGIN
              tell $\text{SUB}_g$ that $Q'$ is satisfactory;
              tell $\text{SUB}_A$ that $Q''$ is satisfactory
            END
        END
    END
END;
fail
END

A while statement. A subordinate whose program segment is a while statement follows the most complicated procedure for generating weakest preconditions, due to its responsibility for making use of alternate invariants when the programmer-specified ones are inadequate. In order to make it easier to understand the behavior of such a
subordinate, it is convenient to temporarily ignore the possibility of generating alternate invariants. For now, consider \texttt{SUB}_{\texttt{while}} to be a subordinate whose task is to find \[\text{[lemma J assert I while B do S]}Q,\] and who lacks the ability to generate invariants other than the programmer-specified ones.

Recall that the weakest precondition function for the \texttt{while} statement is defined:

\[
\text{[lemma J assert I while B do S]}Q =
\]

\[
\text{IF J}&\text{I}&\text{B} \rightarrow Q \text{ is provable}
\]

\[
\text{THEN IF J}&\text{I}&\text{B} \rightarrow [S]I
\]

\[
\text{THEN I}
\]

\[
\text{ELSE FALSE}
\]

\[
\text{ELSE FALSE}
\]

If the loop assertion is not adequate to prove the post-condition \(Q\) or is not inductive, then according to the definition above, the weakest precondition for the statement is \texttt{FALSE}. In intuitive terms, this means that the program in which the \texttt{while} statement appears cannot be successfully verified unless the loop is never executed at all. Assuming that the loop is sometimes executed, \texttt{FALSE} as a weakest precondition is equivalent to there being no precondition that can lead to a successful verification. With this in mind, the behavior of \texttt{SUB}_{\texttt{while}} can be described with the following algorithm:
BEGIN
  IF J\&I\&-B \to Q is provable THEN
  BEGIN
    hire a subordinate SUB_S to find [S]I;
    FIRST precondition I' returned by SUB_S
    SUCH THAT J\&I\&B \to I' is provable
    DO BEGIN
      succeed, returning I;
      IF my superior finds I satisfactory THEN
      BEGIN
        conjoin I to the formula on the lemma clause;
        erase the assert clause;
        tell SUB_S that I' is satisfactory
      END
    END
    OTHERWISE do nothing
  END;
  fail
END

Now, giving SUB_while the ability to make use of alternate invariants is a straightforward matter. Assume that alternate invariants can be generated by appealing to an invariant generator G, which uses heuristic methods to synthesize trial invariants. Then whenever SUB_while discovers that the present asserted invariant is inadequate -- i.e., the invariant either is too weak to prove the exit condition, is not inductive, or is unsatisfactory to SUB_while's superior -- SUB_while requests an alternate invariant from G. If G has no alternate invariants to offer, then SUB_while must fail. The algorithm given above can be modified to accept alternate invariants as follows:
BEGIN
set up a generator G of alternate invariants that subsume the initial asserted invariant I;

REPEAT
  IF J&I&B -> Q is provable THEN
    ...(as above)...;
  request an alternate invariant I from G
UNTIL G offers no more alternate invariants;
fail
END

It is important that verification of the program imply validation of all assertions as initially specified. Thus, G is restricted to generating only invariants that subsume the original asserted invariant.

The above algorithm can be further improved. Recall the discussion in the previous chapter concerning non-inductive loop assertions. There it was shown that it is often possible to find another invariant that can be conjoined to the non-inductive loop assertion to make the loop assertion inductive. This suggests that the SUCHTHAT condition in the above algorithm, i.e.,

J&I&B -> I' is provable

can be replaced by a weaker condition,

I is inductive

where I is determined to be inductive as follows:
IF J & I & B -> I'
THEN I is inductive
ELSE BEGIN
set up a generator H of alternate invariants;
FIRST formula K returned by H
SUCH THAT J & I & B & K -> I' and
K is invariant
DO I is inductive
OTHERWISE I is not inductive
END

As explained in the previous chapter, the invariance of K can be determined through a recursive application of the verifier. For this application, all assert clauses are removed from P before adding the clause "assert K" to SUB while's while statement. If the verification succeeds, then K is known to be a valid invariant. Before continuing the suspended verification, the removed assert clauses must be restored.
A prototype incremental verification system that supports incremental verification and assertion synthesis for programs written in L has been implemented. The system is written in Simula 67 (Dahl, Myhrhaug and Nygaard 1968) and runs interactively on the DECsystem-10. The algorithm used to generate verification conditions is the backtracking algorithm described in the previous chapter. In order to generate alternative inductive assertions, the verifier interacts with the user. The system has successfully used the incremental approach in order to synthesize inductive assertions for a number of small programs, including the binary search program discussed in Chapter 3 and a selection sort.

System Overview

The prototype incremental verification system consists of five components: driver, parser, verifier, trial invariant generator, and theorem prover. Figure 2 gives a schematic diagram of the organization of these components.

Upon invoking the prototype incremental verification system, the user is put in communication with the driver component of the verification system, which immediately
Figure 2. Organization of the Prototype Verification System
requests from the user the file name of the program to be verified. Upon receiving the user's response, the driver invokes the parser to obtain a parse tree for the program stored on the input file. If the input program is syntactically correct, the driver passes the parse tree to the verifier; otherwise, the driver halts, returning the user to the DECsystem-10 monitor. (The verifier, which is for the most part simply an implementation of the algorithm described in Chapter 4, operates on parse trees rather than on a textual representation of the program.) If the verifier succeeds in validating the assertions in the program, it modifies the program's parse tree by deleting the `assert` clauses and making the assertions into lemmas. In this case, i.e., when the verifier succeeds, the driver requests the name of the output file from the user. Then the driver converts the parse tree to a program text, writes the text to the file named by the user, and halts.

For the next pass, the user adds `assert` clauses to the output file using a text editor. Then the user invokes the verification system again, specifying the modified file as the input file for the pass.

It should be emphasized that this implementation is meant to be a prototype implementation, not a production system. Thus, in the Simula 67 program, clarity and easy modification were considered more desirable than space-time
efficiency. Also, the input language accepted by the verification system is a slightly restricted version of L: every arithmetic expression and subexpression formed with a binary operator must be enclosed in parentheses.

Trial Invariant Generator

In the prototype system, the trial invariant generator is the human user. Note that there are two situations in which a trial invariant generator is used, both of which occur during the processing of a while statement by the verifier component:

1. When a loop assertion has been found to be inadequate -- i.e., it either is too weak to prove the loop's postcondition, is not inductive, or is not satisfactory to the superior of the subordinate for the while loop, and

2. When another invariant is needed to make a non-inductive loop assertion inductive.

In the first situation, the verifier displays the loop to the user and the loop's postcondition. Then the user is requested to suggest a plausible invariant. In the second situation, the verifier displays the loop and presents the user with the following message:

I is invariant if R is, where
\[ R \rightarrow (J \land I \land B \rightarrow I') \]

where R stands for the required invariant. J is the verified invariant for the loop, I the asserted invariant, B the
loop condition, and I's a weakest precondition for the loop body given I as the postcondition. Then the verifier requests a trial invariant from the user.

It is easy to see that automatic or semi-automatic heuristic methods for assertion synthesis could be used to replace the user as a generator of trial invariants. The information needed by an automated method for generating trial loop invariants is the same as the information the user obtains from the verifier and upon which the user bases the choice of trial loop invariant.

To facilitate interaction with the user, the while statement recognized by the prototype system has been modified to require a loop identifier. The syntax for this while statement is:

```
lemma <formula> assert <formula>
while ( <identifier> ) <boolexp> do <stmt>
```

where the lemma and assert clauses are optional, as before. The advantage of the new syntax is that it provides each while statement with an identifying name, which the verifier can display to the user in lieu of the loop body when a trial invariant is being requested. This helps prevent the user from being drowned in excessive output.

Theorem Prover

The theorem prover component of the prototype incremental verification system consists of an interactive formula simplifier, which is referred to below as IFS.
Because of the verification system's role as an assertion synthesizer, non-theorems are given to the theorem prover for proof as a matter of course. Thus, it is important that the theorem prover be able to deal effectively with non-theorems as well as theorems. In addition to not wasting inordinate amounts of machine resources in trying to prove non-theorems, it is desirable that the attempted proof of a non-theorem provide a clue for another, better trial invariant.

A formula simplifier deals as effectively with non-theorems as it deals with theorems. A simplifier reduces any formula to a simpler form and then halts. (If the formula is a theorem, then its fully reduced form is TRUE.) Thus, even for non-theorems, a simplifier does not thrash about, searching for a proof. Moreover, the reduced form of a non-theorem may provide the information needed to generated a valid loop invariant. This latter claim is supported in the last section of this chapter.

In the design of the simplifier, it was considered important that it support user-defined predicates. A user-defined predicate is an n-ary predicate function whose properties are given by user-specified axioms and lemmas. An example of a user-defined predicate is:

\[ \text{isin}(x,a[i:j]) \]

which, in intuitive terms, might be defined to mean that the
value of $x$ occurs as the value of one of the elements in the subarray of $a$ consisting of elements $a[i], \ldots, a[j]$. The following axiom partially defines $isin$:

$$isin(x, a[i:j]) \iff x = a[i]$$

The Stanford University Pascal verifier has already demonstrated the desirability of supporting user-defined predicates in a program verification system (Suzuki 1975). One advantage of user-defined predicates is an improvement in expressibility for assertions. With predicates of the user's own invention, assertions can be expressed with shorter formulas and in more intuitive terms. Also, when the user provides lemmas for new predicates, the lemmas can be assumed to be true, or at least their proofs can be deferred until some other time. Thus, proofs of formulas involving these predicates become simpler, i.e., shorter and more perspicuous.

Theorem proving is sufficiently difficult that it can be considered mandatory that a practical theorem prover for a verification system take advantage of human insight (Deutsch 1969; Good et al. 1975; Topor 1975). For this reason, IFS is interactive. After simplifying a formula as much as possible using available information, IFS displays the reduced formula to the user and gives the user the opportunity to provide the simplifier with further
information. Often the simplified form will suggest to the user some lemmas that are sufficient to prove the formula, or at least to simplify it further. If the user chooses to provide additional information of this sort, the simplifier again reduces the formula, armed with the new information. IFS is described in greater detail in the next section.

An Interactive Formula Simplifier

IFS consists of two components: a conditional evaluator and a user interface. The conditional evaluator part is modelled on CEVAL, the formula simplifier used in the ISI verifier for abstract data types (Guttag et al. 1976). The user interface part provides support for user-defined predicates by permitting the user to communicate axioms and/or lemmas to the conditional evaluator. Also, since the proof theory used by the conditional evaluator is incomplete with respect to order relations, equality, and arithmetic operations, the user interface part serves to simulate completion of the proof theory.

The formulas simplified by IFS are the formulas in the assertion logic AL, which is described in Appendix B. Thus, IFS is a simplifier for an applied first-order predicate calculus with equality and without explicit quantifiers.
Conditional Evaluator

The conditional evaluator simplifies formulas in AL by applying rewrite rules until no more apply. The rewrite rules fall into three disjoint sets: decoding rules, reduction rules, and encoding rules. The conditional evaluator first applies the decoding rules until no more apply, then the reduction rules, then the encoding rules.

Rewrite rules are of the form:

\[ R \Rightarrow Q \]

where \( R \) and \( Q \) are formula schemas. Applying such a rewrite rule to a formula \( P \) consists of replacing all occurrences of \( R \) in \( P \) by \( Q \).

**Decoding Rules.** A formula in AL is in **canonical form** if none of the abbreviations given in Appendix B are used. The decoding rules transform AL formulas into canonical form. There is exactly one decoding rule for each abbreviation defined in Appendix B. For example, since \((P \rightarrow Q)\) is an abbreviation for \((\text{if } P \text{ then } Q \text{ else } \text{TRUE})\), the decoding rules include the rule:

\[ (P \rightarrow Q) \Rightarrow (\text{if } P \text{ then } Q \text{ else } \text{TRUE}) \]

Likewise, since \((P \& Q)\) is an abbreviation for \((\text{if } P \text{ then } Q \text{ else } \text{FALSE})\), the following rule is also included in the decoding rules:

\[ (P \& Q) \Rightarrow (\text{if } P \text{ then } Q \text{ else } \text{FALSE}) \]

Thus, applying the decoding rules to the formula
(p \rightarrow (p \& q))
yields the canonical form

\[
\text{if } p \text{ then (if } p \text{ then } q \text{ else FALSE) else TRUE) }
\]

Reduction Rules. The reduction rules simplify a formula in canonical form by removing redundant subexpressions, etc. The reduction rules used by the conditional evaluator are exactly those rewrite rules given for the proof theory of AL in Appendix B. Applying the reduction rules exhaustively to the example formula above yields:

\[
\text{if } p \text{ then } q \text{ else TRUE) }
\]

Encoding Rules. Formulas in canonical form are often unreadable. Since IFS has an interactive component, it is important that formulas be presented to the user in as readable a form as possible. The encoding rules transform a formula in canonical form into a formula that makes use of the abbreviations presented in Appendix B. Thus, the encoding rules rewrite the above formula as:

\[
(p \rightarrow q)
\]
User Interface

The user interface part (UIP) is an integral part of IFS. Although any formula that IFS simplifies to TRUE is true, IFS is not able to prove all true formulas. In particular, the rewrite rules for equality and the ordering relations are not always sufficient for proofs of formulas involving these relations. Also, there are no rewrite rules given that relate to the arithmetic operators. By obtaining additional information from the user, however, IFS is able to prove as many true formulas as necessary for a given verification. Moreover, in addition to completing the rewrite rules with respect to equality, ordering relations and the arithmetic operations, the UIP supports user-defined predicates by providing the user with a way to inform IFS of the properties to be attributed to the predicates.

When the verifier gives IFS a formula to simplify, it gives the formula to the UIP. The UIP then displays the formula to the user and waits for the user to respond, prompting with the symbol #. The user may specify that the UIP perform some symbol manipulation on the formula or request that simplification of the formula be terminated. In the latter case, the UIP returns the formula in its present form to the verifier and relinquishes control. Otherwise, the UIP performs the specified action and prompts the user for the next command.
Note that the conditional evaluator part of IFS is never invoked except at the request of the user. This prevents machine resources from being wasted in attempting to prove formulas that the user recognizes immediately as non-theorems.

The following commands are available to the user: reduce, set, display, terminate, and axiom.

Reduce. The reduce command, which takes no arguments, invokes the conditional evaluator. After the formula has been simplified, the reduced form is displayed to the user.

Set. Let P be the formula being simplified. The syntax for the set command is:

\[ \text{set } x = t \]

where \( x \) is an identifier not occurring in \( P \), and \( t \) is a term that does occur in \( P \). The effect of the command is to replace \( P \) by a new formula

\[ (x \rightarrow Q) \]

where \( Q \) is the formula obtained from \( P \) by replacing all occurrences of \( t \) by \( x \).

Display. The display command, which takes no arguments, displays the formula in its present reduced form to the user.

Terminate. In order to terminate the simplification of the formula, the user responds to the UIP with an empty
line. This signals the UIP to return the formula in its current reduced form to the verifier and to relinquish control.

Axiom. When the conditional evaluator is unable to reduce a formula to TRUE without additional information, the user may help by giving one or more axiom commands. In particular, the user uses axiom commands to define the properties of user-defined predicates.

Each axiom command informs IFS of a formula which the user believes to be true. IFS incorporates this knowledge by constructing a rewrite rule from the given axiom command and applying it to the formula that IFS is currently simplifying. IFS has a very short memory, however; if the axiom is required again later, the user must restate the axiom command.

There are six different forms of the axiom command. Let P, Q, and R be formulas, and s and t be terms. Then the axiom commands are:

\[
\begin{align*}
\text{axiom } s & \text{ is } t \\
\text{axiom if } P & \text{ then } s \text{ is } t \\
\text{axiom } P & \text{ iff } Q \\
\text{axiom if } R & \text{ then } P \text{ iff } Q \\
\text{axiom } P & \text{ if } Q \\
\text{axiom if } R & \text{ then } P \text{ if } Q \\
\end{align*}
\]

The "meanings" of the above commands should be obvious. For example, a command of the form

\[
\text{axiom } s \text{ is } t
\]
means that \( s=t \) is true. The rewrite rule applied by IFS in response to this command is

\[
s \Rightarrow t
\]

Appendix E provides formal meanings for the *axiom* commands and justifies the rewrite rules applied by IFS with respect to these meanings.

A "conditional" *axiom* command (the second, fourth, and last commands above) must be used with care, for a misphrased axiom may be grossly inefficient. At best, it complicates the formula, rather than simplifying it; at worst, it causes the formula to grow too large to be dealt with further. A rule of thumb is to never specify a conditional axiom command in which the formula immediately following the (first) *if* does not appear in the formula being simplified.

Example

The following is an example of how IFS may be used to prove a formula. The formula of the example makes use of the user-defined predicate `isin`, which, the reader will recall, is true if the integer specified by its first argument occurs as the value of one of the elements in the array specified by its second argument.

Below, IFS initiates a dialogue with the user by displaying the formula to be simplified and issuing the prompt symbol \( \# \). The user sees that the formula is rather
redundant, and requests that the formula be simplified as much as possible.

\[
(-(b \neq c) \land 
\neg (isin(key,a[b0:c0]) 
=> 
isin(key,a[((b+c)/2):c]))) 
=> 
(isin(key,a[b0:c0]) 
=> 
key=a[((b+c)/2)])
\]

IFS then simplifies the formula as requested and displays the reduced form to the user. The user notes that removing the multiple occurrences of the term \((b+c)/2\) will make the formula more readable, and so issues a set command to get rid of the multiple occurrences. Then the user displays the resulting formula.

\[
((b=c \land 
(isin(key,a[b0:c0]) \land 
(isin(key,a[((b+c)/2):((b+c)/2)])))) 
=> 
key=a[((b+c)/2)])
\]

Upon reading the displayed formula, the user realizes that the formula is true due to the axiom for isin cited earlier. Thus, the user issues an appropriate axiom command, and requests the formula to be simplified further.
\[ d = \left( \frac{(b+c)}{2} \right) \]

\[ \rightarrow \]

\[ (b=c \land (\text{isin(key, a[b:0:c0])} \land (\text{isin(key, a[d:d]})) \rightarrow \]

\[ \text{key} = \text{a[d]} \]

\# axiom key = a[d] iff isin(key, a[d:d])

\# reduce

This time, IFS is able to reduce the formula to TRUE. Seeing that the formula has effectively been proved, the user terminates the dialogue by responding with an empty line.

TRUE

#

Discussion

IFS was originally designed merely to provide the prototype incremental verification system with its required theorem prover. Although IFS is far from being the ideal theorem prover for program verification, it does have some good attributes that should not be overlooked.

One desirable quality of a program verification system is that it provide documented and precise proofs for the verification conditions. This is especially true when the verifications involve user-defined predicates. A formal proof of a theorem, however, can be very long and tedious, due largely to uninteresting and routine formula manipulation. On the other hand, there are usually one or two interesting facts upon which the entire proof hinges and
which must be documented if the proof is to be documented at all. In fact, in such cases, documentation of the proof can be identified with documentation of these interesting facts.

This is exactly the kind of documentation IFS provides. In the transcript of a dialogue with IFS, all evidence of which builtin rewrite rules were applied by IFS is suppressed. The "interesting" facts needed to complete the proof, however, are visible in the transcript as axiom commands issued by the user.

In order for a verification system to be deemed practical by users, it must contain a degree of informality. For example, when a user introduces a user-defined predicate, the user probably has an intuitive understanding of what that predicate means without fully understanding how to axiomatize its properties. IFS allows the user to be informal to the extent that whenever the user presents an axiom, the axiom is assumed true without question. Since a record of the axiom is retained in the transcript, the user may derive the truth of the axiom at a later time if necessary. Thus, there is no need for the properties of user-defined predicates to be fully axiomatized before verification of a program begins. After all, the user may not even discover the need for inventing a particular predicate until asked to provide a trial invariant.
Factorial Example

In Chapter 3, a program that computes the factorial of a positive integer was used to show the role that loop exit assertions play in incremental verification. The second pass of that example is repeated below, this time with a transcript produced by interacting with the prototype incremental verification system. As before, the factorial function (!) is represented in the example with functional notation, e.g., fact(n), since the assertion syntax is strictly limited by what the prototype system will recognize. Comments on the transcript are presented as paragraphs, while the transcript itself is indented.

The second pass verifies that when the program terminates, \( x = \text{fact}(n) \). The user has stored the asserted program on the file fact2. Immediately upon receiving the input file specification from the user, the driver component of the verification system displays the input program.

```plaintext
input: fact2

entry 0 \leq n;
\begin{align*}
i &:= 0; \\
x &:= 1;
\end{align*}
lemma \ i < n
while(loop) \ i < n do
\begin{align*}
i &:= (i + 1); \\
x &:= (x \times i)
\end{align*}
end;
lemma \ i = n
assert \ x = \text{fact}(n)
```

Since the user has omitted the \texttt{assert} clause from the
program's `while` statement, the asserted invariant defaults to TRUE. Thus, the verifier generates the following verification condition, which if true, shows that the asserted invariant is strong enough to prove the loop's exit assertion. After reducing the formula to a simpler form, the user sees that the formula is not true and terminates the simplification.

```latex
((- (i < n) &
  (i < n &
   TRUE))
 ->
  (i = n
   ->
    x = fact(i)))
# reduce

((n < i &
  n = i)
 ->
  x = fact(n))
#
```

When IFS fails to simplify the verification condition just generated to TRUE, the verifier recognizes that the verification condition is a non-theorem and that a different asserted invariant is needed. The verifier requests a trial invariant from the user, displaying the `while` statement and its exit assertion. (The exit assertion has been subjected to some automatic simplification before being displayed to the user. This explains why the factorial function is applied to `i` rather than `n` in the exit assertion.) Upon receiving the user's trial invariant, the verifier checks
that it subsumes the asserted invariant originally specified in the program text.

```
lemma i < n
while(loop) i < n do ...;
assert (i = n
       ->
       x = fact(i))
```

```
trial invariant: x=fact(i)
```

```
(x = fact(i)
 ->
   TRUE)
# reduce
TRUE
#
```

Having obtained a trial invariant from the user, the verifier generates the following verification condition. If true, this formula shows that the trial invariant is strong enough to prove that when the loop terminates, x=fact(n) is true. The user sees that this verification condition, unlike the previous one, is true and requests that IFS reduce the formula.

```
((~(i < n) &
  (i < n &
   x = fact(i)))
 ->
   (i = n
    ->
    x = fact(i)))
# reduce
TRUE
#
```

When IFS returns TRUE as the reduced form of the verification condition, the verifier next determines whether the trial invariant is inductive. Thus, the verifier generates
the following verification condition, which IFS successfully reduces to TRUE. The user aids in the simplication of the formula by informing IFS of an axiom.

\[ ((i < n \land i < n \land x = \text{fact}(i)) \implies (x \times (i + 1)) = \text{fact}((i + 1))) \]

# axiom if x=fact(i) then (x*(i+1)) is fact((i+1))
# reduce
TRUE
#

Having succeeded in showing that the trial invariant both is strong enough to prove the loop's exit assertion and is inductive, the verifier uses the trial invariant as the weakest precondition for the while statement. The next verification condition generated ensures that the program's entry condition is strong enough to prove that the loop's trial invariant (i.e., precondition) is true at entry to the loop. IFS finds this verification condition to be true.

\[ (0 < n \implies 1 = \text{fact}(0)) \]

# axiom fact(0) is 1
# reduce
TRUE
#

Since the verifier has no more verification conditions to generate, this verifies the program. Thus the verifier modifies the input program by removing the assert clause and adding lemma clauses. Upon receiving the name of an output
file from the user, the driver writes the modified program to the file and halts.

Assertions verified. 
output: fact3

The program on file fact3 is the following:

```plaintext
entry 0 \leq n;
i := 0;
x := 1;
lemma (i \leq n \& x = \text{fact}(i))
while(loop) i < n do
begin
  i := (i + 1);
x := (x \times i)
end;
lemma (i = n \& x = \text{fact}(n))
```

**Insights from Experience**

The prototype incremental verifier has been applied to a number of small programs. The most substantial of these are the binary search program, which was first discussed in Chapter 3, a selection sort program, and a minimal spanning tree program. Listings of these programs appear in Appendix F, together with some transcripts which are the product of interacting with the prototype verification system.

Although the prototype system has been applied to relatively few example programs, even this limited experience has led to some insights regarding the use of an incremental approach to verification.
Recursion and Trial Invariants

In Chapter 3, it was pointed out that it is convenient to be able to call the incremental verifier recursively. Such a recursive call is useful when a trial invariant I for a while statement is not inductive. The idea is to find another invariant K which, when conjoined to I, makes I inductive. The verifier is called recursively to verify that K actually is an invariant.

The backtracking algorithm, which was presented in Chapter 4 and which underlies the incremental verifier, permits recursion when a non-inductive invariant is encountered. Experience with the verifier suggests that this facility is one of the more valuable ones for synthesizing assertions. Incremental verification of a program with nested loops is almost certain to use the recursive facility at some point in the verification. Even verifying single-loop programs sometimes makes use of recursion, as in the binary search example.

What is especially interesting about using the recursive facility of the verifier is how easy it often is to determine what trial invariant needs to be validated in the recursive call. For example, consider the binary search program. (The transcript for this program appears in Appendix F.) The trial invariant first chosen by the user for the while statement is:
(isin(key, a[b0:c0]) -> isin(key, a[b:c]))

where the meaning of isin is given by:

isin(x, y) =
    IF x is an integer and
    y is an array
    THEN IF x occurs as the value of one of
    the elements of y
    THEN true
    ELSE false
    ELSE false

If the following verification condition is true, then the trial invariant is inductive.

((b ⊗ c &
  (TRUE &
   (isin(key, a[b0:c0])
    ->
    isin(key, a[b:c])))
  ->
  (if key > a[((b + c) / 2)]
    then(isin(key, a[b0:c0])
    ->
    isin(key, a[((b + c) / 2) + 1]:c))
    else(isin(key, a[b0:c0])
    ->
    isin(key, a[b:((b + c) / 2)]))))

Since the above formula is so large and full of redundant expressions, it is not clear whether the formula is true or not. Thus, the user reduces the formula as far as possible. The user sets a new variable z equal to the term ((b+c)/2) and applies the axiom

axiom if isin(key, a[b:c]) then

  (if key≤a[z]
    then isin(key, a[b:z])
    else isin(key, a[(z+1):c]))

if (isseted(a[b:c]) & (b≤z & z≤c))

where isseted is intended to have the following meaning:
issorted(x) =
    IF x is an array
    THEN IF the elements of x are in ascending order
        THEN true
        ELSE false
    ELSE false

The verification condition reduces to:

\[
((z = ((b + c) / 2) \&
\begin{array}{c}
  b \neq c \& \\
  (isin(key, a[b0:c0]) \& \\
  isin(key, a[b:c]))
\end{array}
\rightarrow
  (issorted(a[b:c]) \& \\
  (b \leq z \& \\
  z \leq c)))
\]

Even though this verification condition cannot be reduced to TRUE, the trial invariant may still be shown to be inductive, as discussed in Chapter 3. A formula K needs to be found such that

1. K implies the reduced form of the verification condition, and
2. K can be proved to be an actual invariant.

Since the reduced form of the verification condition is an implication, the first condition on K can be satisfied by choosing K to be the consequent of the verification condition, i.e.,

\[
(issorted(a[b:c]) \& \\
(b \leq z \& \\
  z \leq c))
\]

Since z does not appear in the program, a better choice of K is obtained by removing z from the above formula, i.e.,
As the transcript in Appendix F shows, K satisfies both of the necessary conditions, so that recursively calling the verifier to validate K as an invariant succeeds.

A similar situation occurs in the selection sort program. In this example, the following user-defined predicate is introduced:

\[
\text{part}(x, y) = \\
\text{IF } x \text{ is an array and } y \text{ is an integer} \\
\text{THEN IF each element of } x[x_{\lfloor b : y \rfloor}] \\
\text{is less than or equal to each element} \\
in x[(y+1):x_{ub}] \\
\text{THEN true} \\
\text{ELSE false} \\
\text{ELSE false}
\]

In verifying the selection sort example, during the fourth pass through the incremental verification system (see Appendix F), the user introduces the formula

\[\text{issorted}(a[1:i])\]

as a trial invariant for the outer loop. This trial invariant is inductive if the following verification condition is true:

\[
((i \leq n \& \\
(i \leq n \& \\
\text{issorted}(a[1:i]))) \\
\rightarrow \\
(\text{part}(a[1:n],((i + 1) - 1)) \& \\
\text{issorted}(a[1:((i + 1) - 1)])))
\]

Applying the axiom
axiom \(((i+l)-1)\) is \(i\)

the user reduces this formula to:

\[
((i < n \&
  \text{issorted}(a[1:i]))
\rightarrow
  \text{part}(a[1:n], i))
\]

The formula cannot be reduced further. Nevertheless, the trial invariant can be shown to be inductive by finding a formula \(K\) that is invariant and implies the reduced form of the verification condition. As in the binary search example, the verification condition is an implication, which suggests the consequent of the implication as a plausible choice for \(K\). The transcript shows that this choice of \(K\) is satisfactory.

These examples further justify implementing the theorem prover component of the incremental verification system with a formula simplifier. Suppose a trial invariant cannot be shown to be inductive because the verification condition that expresses its inductiveness cannot be proved. Then a recursive application of the verifier may yet be used to show the inductiveness of the trial invariant, if another suitable invariant can be found. It has just been demonstrated that the reduced form of the unprovable verification condition can be used to find such a suitable invariant. Thus, the formula simplifier is useful in its role as a simplifier as well as a theorem prover. Before deriving the
needed invariant from the unprovable verification condition, it is important that the verification condition be reduced as far as possible. This ensures that the derived invariant is no stronger than it need be to make the original invariant inductive.

Expressing Properties of Loops

Chapter 3 pointed out how incremental verification can be used advantageously to verify programs containing multiple loops. In particular, separate passes may be used to verify that each loop does what it is expected to do.

The statement of a loop's expected effect can be expressed from two points of view, however. From one point of view, the loop is part of a larger program, and its effect contributes to the correctness of the whole program. From another point of view, the loop is an entity that is more or less independent of the rest of the program; its effect can be expressed without regard for what the program as a whole is supposed to do. Intuition suggests that if the incremental approach is used, the statement of a loop's effect is best expressed according to the second point of view, i.e., in terms that are independent of the program considered as a whole; experience with the prototype verification system supports this intuition. Verifying the inner loop of the minimal spanning tree program (see Appendix F)
illustrates what may happen when the effect of a loop is expressed from each of these two different points of view.

The algorithm used in the minimal spanning tree program is essentially the one described in Aho, Hopcroft and Ullman (1976), except that it uses single-dimensioned arrays to simulate the sets of the Aho et al. algorithm. Briefly, the program works by transforming a spanning forest for the graph in question into a single spanning tree. At the start of the program, the forest consists entirely of single-vertex trees; at the end of the program, the forest consists of exactly one tree, which spans the entire graph. The vertices of the graph are numbered from 1 to m. The array heap[1:m] assigns each vertex to one of the trees in the forest: if heap[i]=heap[j], then vertices i and j occur in the same tree in the spanning forest; otherwise, they occur in different trees. Thus, the number of different values in heap[1:m] is the same as the number of trees in the spanning forest.

A reasonable way to start verifying the spanning tree program is to verify that upon exiting the outer loop, there is exactly one tree in the forest. However, in order to take full advantage of the incremental approach, the inner loop should be verified before the outer loop. The effect of the inner loop can be expressed in different ways, depending on what point of view is taken for the loop. If
the inner loop is seen as part of the larger program, then
the inner loop's exit assertion might be expressed as "the
number of trees in the spanning forest is one less than the
number of trees that were in the forest before executing the
loop". On the other hand, if the inner loop is considered
independently of its role in the larger program, then the
exit assertion becomes "heap[1:m] is the same as it was
before the loop, except that all elements with value t have
been replaced by h".

The prototype system was used to verify the inner
loop with respect to formalized versions of each of these
exit assertions. The first exit assertion turned out to be
very awkward to prove, and a successful verification never
was achieved. The proof of the second exit assertion,
however, was relatively straightforward, and led eventually
to verifying the outer loop with respect to the number of
trees in the spanning forest. It appears that the second
form of the exit assertion is easier to prove because it
expresses a "first-order" effect of the loop. The other
form of the exit assertion, which views the loop as part of
the larger program, expresses a "second-order" effect.
Since this second-order effect depends on other properties
of the program as well as the first-order effect, its proof
is more difficult to obtain.
Summary

Applying the prototype incremental verification system to a few examples has supported the following conclusions:

1. The ability to call the verifier recursively is an especially valuable aspect of the verification system.

2. The reduced form of a formula that expresses the condition that an invariant be inductive may be used to derive another invariant which can be conjoined to the original invariant to make it inductive.

3. When verifying inner loops, it is best to give the loop's exit assertion in terms that do not relate to the program as a whole, but rather express what the loop does independently of the rest of the program.
CHAPTER 6

CONCLUSIONS

Summary

This dissertation has presented an incremental approach to verification that provides a natural and intuitive way of partitioning a correctness proof into a number of smaller, simpler proofs. Using this approach, a typical verification consists of several passes through a program verifier. Each successful pass verifies more properties of the program, and in the process validates another set of assertions for the program. Assertions that have been validated in previous passes are recognized in subsequent passes as facts, or lemmas, which can be drawn upon to synthesize and/or validate other assertions. During a verification, if the inductive assertions supplied with the program are not sufficient to complete the verification, new inductive assertions are generated as they are needed.

Incremental verification requires that programs be able to express their own properties. This dissertation defines a simple, Algol-like programming language L, which supports incremental verification. In L, program properties are expressed in assert and lemma clauses. During a
verification, assertions are proved true, while lemmas are assumed true. A successful pass validates all assertions appearing in the program and writes a new program exactly like the original, except that all validated assertions appear in the new program as lemmas.

The incremental verifier is essentially a controller for heuristic techniques for assertion synthesis. When an inductive assertion is found to be inadequate, the verifier backtracks, uses any available heuristic techniques to generate an alternate inductive assertion, and continues with the verification. A product of this research is an algorithm for an incremental verifier. The algorithm processes programs in L in a top-down fashion, and is based on the computation of weakest preconditions. Although L has a very limited number of iterative control structures, the algorithm can easily be extended to additional iterative control structures.

A prototype incremental verification system has been implemented in Simula 67 and has successfully verified a number of small programs. The system consists of a driver (main program), parser, verifier, trial invariant generator, and theorem prover. The parser accepts programs written in L, the verifier is a coroutine-based implementation of the backtracking algorithm mentioned above, the trial invariant generator is the human user, and the theorem prover is an interactive formula simplifier called IFS.
IFS is based upon two earlier formula simplifiers. Its method for reducing formulas is virtually identical to the conditional evaluator of the ISI verifier for abstract datatypes (Guttag et al. 1976). Its facilities for user-defined predicates were suggested by similar facilities of the Stanford Pascal verifier (Igarashi et al. 1975). In addition, IFS has an original interactive component.

Contributions

The major contributions of this research lie in the incremental nature of the proof method, the step-wise development of inductive assertions, some improvement in the interface between humans and automatic program verification, and an algorithm for applying heuristic techniques for assertion synthesis.

Incremental Nature of the Proof Method

The significance of the incremental proof method presented in this dissertation is that it improves upon the practicality of using the method of inductive assertions for program verification. Each pass through the verifier generates verification conditions that are shorter and/or easier to prove than those of a one-pass approach. Moreover, the inductive assertions required in each pass are shorter and easier to derive. Thus, the incremental approach alleviates two of the most difficult problems associated with the
method of inductive assertions: theorem proving and con­structing the required inductive assertions. This is the result of two features of the incremental approach.

First, lemmas are recognized as proven program properties, and are used to help validate and synthesize other assertions. When lemmas appear in a program, the verification conditions generated for the program are often easier to prove, because the lemmas contribute information that can be assumed true. Lemmas also provide information from which inductive assertions can be derived. Moreover, the presence of lemmas in a program means that validation of the program's assertions may be accomplished with shorter inductive assertions.

Second, the user initiates each pass by augmenting the program being verified with new assertions. Because validated assertions become lemmas for subsequent passes, this means that the user strongly influences the course of the verification.

Note that the incremental approach is made possible by the existence of a lemma statement in the verification language, which specifies assertions that can be assumed true. The prototype verifier is not the first verifier to include such a statement in its verification language; the language recognized by Deutsch's verifier includes an ASSUME statement, which has a similar meaning (Deutsch 1969). The
value of the ASSUME statement for incrementally verifying programs, however, was not realized or exploited in Deutsch's work.

It is illustrative to compare the incremental verification of a program with more conventional approaches which do not recognize valid assertions. For simplicity, assume that the backtracking facility for generating alternative inductive assertions has been omitted from the incremental approach. Now consider applying a conventional verifier that is based on the method of inductive assertions to a "correct" program. If the program's inductive assertions are inadequate — i.e., either incorrect or incomplete or both — then the verifier puts out a number of simplified verification conditions, not all of which are TRUE. After examining this output, the user modifies some of the inductive assertions in the program and tries again (VonHenke and Luckham 1974). It is easy to see that it may take many attempts before the user succeeds in supplying a complete set of correct inductive assertions; each modification raises the possibility of introducing new errors into the program's inductive assertions.

In contrast, the incremental approach to program verification serves to partially constrain the modification of inductive assertions. Whenever a pass is successful, its inductive assertions get incorporated as lemmas into the
program. In subsequent passes, these lemmas need never be considered for modification, since they are known to be valid. In a sense, the incremental approach encourages a series of modifications to converge to a complete set of correct inductive assertions.

Another advantage the incremental approach has over conventional verification with inductive assertions appears in the case of incorrect programs. Using separate passes to verify different properties of a program helps isolate the source of error.

Step-Wise Development of Inductive Assertions

In this research, loop assertions are developed by finding the loop invariants one at a time and forming their conjunction. Whenever a loop assertion is found to be inadequate, the assertion is further developed by finding an invariant that, when it is conjoined to the loop assertion, improves the adequacy of the loop assertion. Verifying a program incrementally, i.e., over several passes, is used to gradually develop the inductive assertions necessary to verify the program. The basic idea is that each pass through the verifier finds loop invariants one at a time and conjoins them to the program's loop assertions.

The idea of synthesizing inductive assertions by forming the conjunction of a number of individual loop
invariants is not new; VISTA (German and Wegbreit 1975), for instance, does exactly this. What is original, however, is the goal-directed approach presented in this thesis for finding a set of invariants sufficient to verify a program. Whereas VISTA simply continues collecting invariants until an adequate set is obtained, the approach described here seeks invariants that will improve the adequacy of the inductive assertion. It has been shown that each invariant in an inductive assertion plays a specific role in ensuring the adequacy of the assertion. In particular, each invariant ensures either that the assertion is strong enough to verify an exit condition of the loop or that the assertion is inductive over the loop. These roles direct the search for trial invariants.

The following guidelines for developing loop assertions using the incremental approach to verification have emerged.

1. With separate passes, verify that each loop has its expected effect. If inner loops are verified before outer loops, then the loop assertions on the inner loops are built up gradually, usually without requiring any great leaps of imagination.

2. Verify separate effects of loops separately, first-order effects before second-order ones. Not only does this
allow for the gradual development of inductive assertions, but also the knowledge of a first-order effect may contribute to the generation of plausible trial invariants.

3. When a loop assertion is found not to be inductive, try finding another formula, which when conjoined to the loop assertion, makes it inductive. Then recursively call the verifier to verify that this formula is an invariant.

Nested Loops

In the case of nested loops, the loops' inductive assertions are quite interdependent. As a result, both algorithmic and heuristic methods for synthesizing assertions tend to flounder for programs containing nested loops. Nevertheless, an incremental approach can be used to ease the task of developing inductive assertions in this case. The interdependencies are controlled by verifying in separate passes that each loop has its expected effect. The statement of a loop's effect simplifies finding needed invariants for the loop as follows.

First, in the case of an inner loop, verifying the loop for its effect results in adding a lemma at the loop exit. In subsequent passes, this lemma may be used (via the
exit heuristic given in Chapter 3) to help synthesize a loop invariant needed to verify an outer loop.

Second, a loop's exit condition provides a starting point from which heuristic methods can synthesize trial invariants. A programmer-specified exit condition is likely to be more usable than one that has been derived in the course of verifying another part of the program, e.g., an enclosing loop. Thus, the incremental approach does not compete with heuristic methods for assertion synthesis, but rather focuses their efforts.

Human-Machine Interface

This research contributes to making a viable human-machine interface for a program verification system, while incorporating some results of previous work in this area. A basic design philosophy underlying the incremental approach to verification and the prototype incremental verification system is that human insight should be used to advantage at every opportunity. In fact, the incremental approach and the prototype system are successful largely because they take advantage of human insight and human interaction.

Human insight is exploited when the user initiates each pass through the verifier by augmenting the program with one or more new assertions. An added assertion need not appear at the end of the program, and more than likely
will appear immediately after a loop to express the expected effect of the loop. So placed, the assertion has a strong influence upon what trial invariants will be synthesized for the loop. Since it often is more intuitive to explain a loop in terms of its effect rather than to provide an invariant property, permitting the user to state the effect of a loop is a good utilization of human insight.

This research incorporates human interaction into the prototype verifier in a way different from any other existing verification system: the prototype system permits the user to attach inductive assertions to a program during verification rather than requiring that the inductive assertions all be provided before verification begins. Whenever the verifier needs an inductive assertion to generate verification conditions, it requests one from the user. If the (user-supplied) trial assertion is subsequently found to be inadequate, the verifier backtracks and requests a different trial assertion from the user. This provides the user with immediate feedback concerning the adequacy of the proposed inductive assertions, and allows the user to try alternate inductive assertions without starting the entire verification over each time a trial assertion is found to be incorrect.
The Backtracking Verification Algorithm

Although a number of heuristic methods for assertion synthesis can be found in the literature, nobody gives a coherent account of how these methods can be incorporated into a program verifier. The problem of how to deal with nested loops has especially been neglected. VISTA (German and Wegbreit 1975) implements some heuristic methods for assertion synthesis, but apparently relegates the backtracking control necessary to test the validity of trial invariants to the user. The value of user interaction in program verification cannot be disputed. Nevertheless, if a suitable algorithm can be found to efficiently perform some task previously relegated to the user, then there is no reason to continue burdening the user with that task. Thus, an important contribution of the present research is the backtracking algorithm presented for the verifier. This algorithm does just the kind of trial invariant testing and backtracking that is required for synthesizing inductive assertions using heuristic methods. The algorithm even handles nested loops, which have always been viewed as difficult problems.

Directions for Future Research

This research has been directed at the problem of developing inductive assertions for iterative programs. There are a number of topics suggested by this research that
warrant further investigation. For the most part, these topics are concerned with integrating the results obtained from this research with other methods for synthesizing inductive assertions. Other topics are wider application of the prototype verification system, the implementation of a more realistic verification system based on the verification method described in this dissertation, and applying verification methods to incorrect programs.

Implementing Heuristic Generation of Invariants

Due largely to the difficulty of controlling the testing of trial invariants, few implementations of heuristic methods for synthesizing inductive assertions have been undertaken. VISTA (German and Wegbreit 1975) includes some heuristic techniques for assertion synthesis, but much of the testing of trial invariants (inductive assertions) is controlled interactively rather than by means of an automated backtracking algorithm. This dissertation, however, has presented a conceptually clean backtracking algorithm with which to control the generation and testing of trial invariants. Thus, an obvious extension of this research is to implement the heuristic techniques for synthesizing inductive assertions — for example, Greif and Waldinger's technique (1974) of propagating a loop's output assertion backwards through the loop to obtain an inductive assertion
and some of the heuristics proposed by Katz and Manna (1976). An implementation effort of this sort is important because it can be expected to lead to further insights into automatic assertion synthesis, as well as to pinpoint any flaws that the proposed methods may contain.

The backtracking verification algorithm presented in this dissertation is not suitable for all heuristic methods, however. The algorithm is based on the computation of weakest preconditions, so that during verification, the program's assertions are essentially pushed backwards through the program. Some heuristic methods, on the other hand, require that assertions be pushed forward through the program to obtain new trial invariants. Although the present algorithm is unsuitable for "forward pushing" heuristic methods, it is straightforward to construct a similar algorithm that is based on strongest postconditions (which are dual to weakest preconditions) and that pushes assertions forward through the program.

Toward a Realistic Verification System

If verification is ever to become a useful tool for programmers, then realistic verification systems must be implemented. Even if program verification using the method of inductive assertions is not destined to be a practical tool, experimental verification of "real" programs is
necessary to further our understanding of what is entailed in the verification of "real" programs. Thus, at a pragmatic level, a more realistic verification system that is based on the results and basic design philosophy of this research needs to be implemented.

A first step toward implementing a more realistic verification system would be to increase the repertoire of iterative control structures that are recognized by the prototype system. For instance, control structures such as \texttt{repeat-until} and \texttt{for} can be integrated into the verifier algorithm quite easily. In the same vein, \texttt{L} needs to be replaced by a real programming language or a subset. An Algol-60 or Pascal subset would be a reasonable choice. In order to be useful, however, whatever subset is chosen should include some kind of procedure mechanism.

The next step is to replace \texttt{IFS} by a less primitive and more efficient formula simplifier. \texttt{IFS} facilities for user-supplied axioms are a clear place for improvement: the formula simplifier should remember the axioms it is given for later use. Another improvement would be to give \texttt{IFS}, or its replacement, the capability of disproving non-theorems. This can be accomplished quite easily by adding an interactive facility for the user to construct counterexamples, similar to the counterexample facility in PIVOT (Deutsch 1969). It would also be desirable to extend the recognized
formulas to include quantified ones. The techniques for instantiating quantified formulas suggested by Wegbreit (1977) suggest that an interactive simplifier could be built that deals with existentially and universally quantified formulas. In order for the verification system to be usable, it is important that the formula simplifier remain interactive. The formula simplifier used by the Stanford Pascal verifier (Igarashi et al. 1975) might be a good replacement for IFS, if suitable interactive facilities were incorporated into it.

Finally, a realistic incremental verification system should be able to invoke some primitive automatic or semi-automatic techniques for generating trial invariants. The user should be given the opportunity of rejecting any mechanically generated invariant, of modifying it, or of replacing it with one of the user's own invention.

Orderly Generation of Trial Invariants

When the verifier discovers that a trial invariant for a loop is not adequate, the verifier backtracks, generates a new trial invariant, and continues the verification with the new invariant. Unless there is some orderly manner in which invariants are generated, however, there is no way of guaranteeing that the new invariant is any closer to being satisfactory. An interesting problem would be to
develop a theory for ordering the sequence of generated invariants. The goal would be to constrain the possible sequences of generated invariants to being convergent while maintaining "completeness". Completeness, in this sense, means that eventually an invariant will be generated that is sufficient for verifying the loop with respect to whatever exit condition is presently under consideration for the loop. This problem is very difficult and the computation of a convergent sequence of trial invariants is already known to be computationally intractable (Wegbreit 1976). Nevertheless, an investigation of the problem is warranted in order to further our understanding of the generation of trial invariants, i.e., inductive assertion synthesis, and to see how close one can come to the ideal while keeping the computation within bounds.

Exploiting Lemmas

Since lemmas, i.e., proven program properties, express information about a program that may not easily be derivable from the original program text, they can be used to help synthesize inductive assertions. For example, VISTA (German and Wegbreit 1975) propagates forward any newly proven program properties to obtain valid assertions elsewhere in the program. Another example is the exit heuristic given in Chapter 3, which is a stronger version of a
heuristic used by VISTA to generate trial invariants; the
exit heuristic, unlike VISTA's weaker one, takes into
account any lemmas that are true at exit from the loop.
Undoubtedly further research will uncover other means of
exploiting proven program properties. The fact that such
program properties can be made conveniently accessible via
*lemma* clauses should facilitate research in this direction.

**Experimentation**

More experimental verification of small programs
needs to be done using the prototype verification system.
Such experimentation can be expected to yield deeper
insights into the internal structure of inductive asser­
tions, and consequently, more heuristics for incrementally
constructing trial invariants.

**Incorrect Programs**

Most of the research in program verification has
been devoted to proving that correct programs are in fact
correct. Although Katz and Manna (1976) show some results
concerning how verification techniques might be used to
systematically uncover program errors, comparatively
little attention has yet been given to this problem.
Although an incremental approach to verification helps iso­
late program errors to some extent, it is far from being
entirely satisfactory. Before program verification can
become a useful tool, the error detection aspect of program verification needs to be vigorously investigated. After all, the majority of programs that will be subjected to verification will be incorrect ones.

**Perspective**

This research was begun with the hope of discovering some new methods for synthesizing inductive assertions. This has been accomplished, although in a rather different way than was originally envisioned. It is the opinion of the author that the most important theoretical contribution of this work is the discovery that inductive assertions are decomposable into separate loop invariants, each of which can be sought by considering some constraint on the inductive assertion. This discovery, together with the incremental proof method developed in this research, furthers the practicality of the method of inductive assertions by permitting a program's proof of correctness to be developed as a number of smaller, simpler proofs.
APPENDIX A

SYNTAX AND AXIOMATIC SEMANTICS FOR L

Syntax

<pgm> ::= entry <formula> ; <stmtlist>
<stmtlist> ::= <stmtlist> ; <stmt>
::= <stmt>
<stmt> ::= <variable> := <exp>
::= let <variable> := <exp>
::= <variable> [ <exp> ] := <exp>
::= begin <stmtlist> end
::= if <boolexp> then <stmt> else <stmt>
::= null
::= lemma <formula> assert <formula>
::= lemma <formula> assert <formula>
::= while <boolexp> do <stmt>
<boolexp> ::= ( if <boolexp>
             then <boolexp> else <boolexp> )
::= ( <boolexp> <lop> <boolexp> )
::= <exp> <relop> <exp>
::= ~ <boolexp>
::= TRUE
::= FALSE
<exp> ::= ( if <boolexp> then <exp> else <exp> )
::= ( <exp> <op> <exp> )
::= <variable> [ <exp> ]
::= <function name> ( <explist> )
::= <variable>
::= <number>
<explist> ::= <explist> , <exp>
::= <exp>
<relop> ::= =
::= ≠
::= <
::= ≤
::= ≥
::= >
<lop> ::= \rightarrow
    ::= \leq
    ::= \&
    ::= \lor

<op> ::= +
    ::= -
    ::= *
    ::= /
    ::= **

<ident> is an identifier, consisting of a letter followed by an arbitrary number of letters and/or digits.

<number> is an arbitrary number of digits, optionally preceded by a minus sign.

<formula> is any <formula> in the assertion language AL (see Appendix B).

The let statement is further constrained: the identifier on the lefthand side of the assignment can appear elsewhere in the program only in assert clauses, lemma clauses, and other let statements.

Lemma and assert clauses may be omitted, except that both clauses may not be omitted in a lemma-assert statement. The formula on such an omitted clause defaults to TRUE.

Pairs of parentheses may be dropped if they are extraneous in light of the following conventions. All the binary logical operators and binary <op>s associate to the left, except for \rightarrow and **, which associate to the right.

The precedence of operators are:

highest: \rightarrow **
next highest: \& or * /
lowest: \leq, \rightarrow + -
Axiomatic Semantics

In keeping with the viewpoint of axiomatic semantics discussed in Chapter 2, the axiomatic semantics for L is presented below as a system of logic, which is referred to as ASL for convenience. The assertion logic for ASL is AL, presented in Appendix B.

Formulas

The formulas of ASL are the formulas of AL, as well as verification statements of the form

\{P\} A \{Q\}

Above, P and Q are formulas in the assertion logic AL and A is a <pgm>, <stmtlist>, or <stmt> in L.

Semantics

A verification statement \{P\}A\{Q\} means: if P is true immediately before executing A and if A terminates, then Q is true immediately after executing A. In both L and AL, the variables range over the integers and arrays of integers. The constants 0, 1, -1, etc. have their usual meaning in the integers.

Proof Theory

The axioms of ASL are exactly the theorems of AL. The primitive rules of inference of ASL are given below. Each rule is grouped with a production from the BNF.
description of L and one of the cases of the definition for
the weakest precondition function for L (see Chapter 4).
This format was chosen to emphasize that
1. The bottom halves of the rules of inference are deter-
mined by the grammar, and
2. The definition of the weakest precondition function is
derived in a straightforward manner from the rules of
inference.

Notation: \( P, Q, R, R_1, R_2, I, J \) are <formula>s in AL.
\( S, S_1, S_2 \) are <stmt>s in L.
\( A \) is a <stmtlist> in L.
\( B \) is a <boolean> in L.
\( x \) is an <ident> in L.
\( fx, i \) are <exp>s in L.

\( Q(x/fx) \) is the formula obtained from \( Q \) by substitut-
ing \( fx \) for all (free) occurrences of \( x \).

\( Q(x[j]/(if \ i=j \ then \ fx \ else \ x[j])) \) is the formula
obtained from \( Q \) by substituting
1. \( (if \ i=j \ then \ fx \ else \ x[j]) \) for all (free) occurrences of
\( x[j] \), for any \( j \), and simultaneously
2. \( x[i<-fx] \) for all (free) unsubscripted occurrences of \( x \).

Note that <exp>s and <boolean>s of L are also
<term>s and <formula>s, respectively, of AL. If this were
not so, some of the rules of inference below would specify
syntactically incorrect formulas.
PR1.  \(<\text{pgm}> ::= \text{entry} \ <\text{formula}> ; \ <\text{stmtlist}>\) 

\[
\{R\} \ A \ {Q} \\
\----------------------------- \\
\{P\} \ \text{entry} \ R; \ A \ {Q}
\]

\[\text{entry} \ R; \ A \ Q = \text{IF} \ R \rightarrow [A] Q \]
\[\text{THEN} \ \text{TRUE} \]
\[\text{ELSE} \ \text{FALSE} \]

PR2.  \(<\text{stmtlist}> ::= <\text{stmtlist}> ; <\text{stmt}>\) 

\[
\{P\} \ A \ {R}, \ {R} \ S \ {Q} \\
\----------------------------- \\
\{P\} \ A; \ S \ {Q}
\]

\([A; \ S]Q = [A][S]Q\)

PR3.  \(<\text{stmt}> ::= <\text{ident}> := <\text{term}>\) 

\[
P \rightarrow \ Q(x/fx) \\
\----------------------------- \\
\{P\} \ x := fx \ {Q}
\]

\([x:=fx]Q = Q(x/fx)\)

PR4.  \(<\text{stmt}> ::= \text{let} <\text{ident}> := <\text{term}>\) 

\[
P \rightarrow \ Q(x/fx) \\
\----------------------------- \\
\{P\} \ \text{let} \ x := fx \ {Q}
\]

\([\text{let} \ x:=fx]Q = Q(x/fx)\)

PR5.  \(<\text{stmt}> ::= <\text{ident}> [ <\text{exp}> ] := \text{fx}\) 

\[
P \rightarrow \ Q(x[j]/(\text{if} \ i=j \ \text{then} \ \text{fx} \ \text{else} \ x[j]))) \\
\----------------------------- \\
\{P\} \ x[i] := \text{fx} \ {Q}
\]

\([x[i] := \text{fx}]Q = Q(x[j]/(\text{if} \ i=j \ \text{then} \ \text{fx} \ \text{else} \ x[j])))\)
PR6. \( <\text{stmt}> ::= \text{begin} <\text{stmtlist}> \text{end} \)
\[
\{P\} \text{A} \{Q\} \\
\text{-------------------} \\
\{P\} \text{begin A end} \{Q\}
\]
\([\text{begin A end}]Q = [\text{A}]Q\)

PR7. \( <\text{stmt}> ::= \text{if} <\text{boolexp}> \text{then} <\text{stmt}> \text{else} <\text{stmt}> \)
\[
\{P&B\} \text{S1} \{Q\}, \{P&-B\} \text{S2} \{Q\} \\
\text{-------------------} \\
\{P\} \text{if B then S1 else S2} \{Q\}
\]
\([\text{if B then S1 else S2}]Q = \)
\((\text{if B then } [\text{S1}]Q \text{ else } [\text{S2}]Q)\)

PR8. \( <\text{stmt}> ::= \text{null} \)
\[
P \rightarrow Q \\
\text{-------------------} \\
\{P\} \text{null} \{Q\}
\]
\([\text{null}]Q = Q\)

PR9. \( <\text{stmt}> ::= \text{lemma} <\text{formula}> \text{assert} <\text{formula}> \)
\[
P&R1 \rightarrow R2&Q \\
\text{-------------------} \\
\{P\} \text{lemma R1 assert R2} \{Q\}
\]
\([\text{lemma R1 assert R2}]Q = R1 \rightarrow R2&Q\)

PR10. \( <\text{stmt}> ::= \text{lemma} <\text{formula}> \text{assert} <\text{formula}> \text{while}... \)
\[
P \rightarrow J, \{I&J&B\} S \{J\}, I&J&-B \rightarrow Q \\
\text{-------------------} \\
\{P\} \text{lemma I assert J while B do S} \{Q\}
\]
\([\text{lemma I assert J while B do S}]Q = \)
\(\text{IF } I&J&-B \rightarrow Q \text{ THEN IF } I&J&B \rightarrow [S]J \text{ THEN J} \text{ ELSE FALSE} \text{ ELSE FALSE} \)
Additional rules of inference for ASL include any rules that are derivable from the axioms and the primitive rules of inference.

A proof in ASL is as defined in Chapter 2. Thus, a proof is a sequence of formulas of AL and/or verification statements, where the last item in the sequence is a verification statement. Each item in the proof is an axiom or a direct consequence of previous items.
APPENDIX B

DESCRIPTION OF THE ASSERTION LOGIC AL

Syntax

<formula> ::= ( if <formula> then <formula> else <formula> )
 ::= <term> = <term>
 ::= <term> < <term>
 ::= <predicate name> ( <termlist> )
 ::= TRUE
 ::= FALSE

<termlist> ::= <termlist> , <term>
 ::= <term>

<term> ::= ( if <formula> then <term> else <term> )
 ::= ( <term> <op> <term> )
 ::= <term> [ <term> ]
 ::= <term> [ <term> <- <term> ]
 ::= <term> [ <term> : <term> ]
 ::= <function name> ( <termlist> )
 ::= <variable>
 ::= <number>

<op> ::= +
 ::= -
 ::= *
 ::= /
 ::= **

<predicate name> ::= <ident>
<function name> ::= <ident>
<variable> ::= <ident>

<ident> is an identifier, consisting of a letter followed by an arbitrary number of letters and/or digits.

<number> is an arbitrary number of digits, optionally preceded by a minus sign.
In addition, the following abbreviations are recognized:

\[
\begin{align*}
(P & Q) & \text{ for } (\text{if } P \text{ then } Q \text{ else } \text{FALSE}) \\
(P \lor Q) & \text{ for } (\text{if } P \text{ then } \text{TRUE} \text{ else } Q) \\
(P \implies Q) & \text{ for } (\text{if } P \text{ then } Q \text{ else } \text{TRUE}) \\
(P \iff Q) & \text{ for } (\text{if } P \text{ then } Q \text{ else if } Q \text{ then } \text{FALSE} \text{ else } \text{TRUE}) \\
\neg P & \text{ for } (\text{if } P \text{ then } \text{FALSE} \text{ else } \text{TRUE}) \\
s \neq t & \text{ for } (\text{if } s = t \text{ then } \text{FALSE} \text{ else } \text{TRUE}) \\
s > t & \text{ for } (\text{if } t < s \text{ then } \text{FALSE} \text{ else } \text{TRUE}) \\
s < t & \text{ for } (\text{if } s < t \text{ then } \text{FALSE} \text{ else } \text{TRUE}) \\
s < t & \text{ for } (\text{if } s > t \text{ then } \text{FALSE} \text{ else } \text{TRUE})
\end{align*}
\]

Pairs of parentheses may be dropped if they are extraneous in light of the conventions given in Appendix A.

**Semantics**

**Notation**

An ordered n-tuple is represented below as a list of its members, enclosed in angular brackets, e.g., 
\(<m_1, \ldots, m_n>\). If \(x\) is an n-tuple then, for all \(i\) such that \((1 < i < n)\), \(x_i\) represents the \(i\)th member of \(x\). Thus, if \(x\) is the n-tuple \(<m_1, \ldots, m_n>\), \(x_i\) represents \(m_i\).

If \(x\) is an n-tuple then, for all \(i\) such that \((1 < i < n)\), \(x(i < y)\) represents the n-tuple \(<x_1, \ldots, x_{i-1}, y, x_{i+1}, x_n>\).

An array is a triple \(<lb, ub, value>\), where \(lb\) and \(ub\) are integers such that \(lb < ub\) and \(value\) is a \((ub - lb + 1)\)-tuple. For readability, if \(x\) is an array, \(x_{lb}\) is used for \(x_1\), \(x_{ub}\) for \(x_2\), and \{x\} for \(x_3\).
The Interpretation Function

The domain of AL consists of the integers ..., -1, 0, 1, ..., an "undefined" value UNDEF, and arrays of integers and/or UNDEF values. Meanings for terms and formulas in AL are given by the interpretation function J. Given that each variable has been assigned a value from the domain, the function J maps each term into an integer, UNDEF, or an array, and maps each formula into truth (T) or falsehood (F).

The function definition for J uses the following conventions:

\[ s, t, i, j, \]
\[ t_1, t_2, ..., t_n \text{ are terms} \]
\[ P, Q, R \text{ are formulas} \]
\[ g \text{ is a function name} \]
\[ p \text{ is a predicate name} \]

The following cases of the definition of the interpretation function map terms into elements in the domain of AL.

\[ J(0) = \text{the integer } 0 \]
\[ J(1) = \text{the integer } 1 \]
\[ J(-1) = \text{the integer } -1 \]
\[ \text{etc.} \]

\[ J(s + t) = \]
\[ \text{IF } J(s) \text{ and } J(t) \text{ are integers} \]
\[ \text{THEN } J(s) + J(t) \]
\[ \text{ELSE UNDEF} \]

\[ J(s - t) = \]
\[ \text{IF } J(s) \text{ and } J(t) \text{ are integers} \]
\[ \text{THEN } J(s) - J(t) \]
\[ \text{ELSE UNDEF} \]
\( J(s\ast t) = \)
   \[
   \text{IF } J(s) \text{ and } J(t) \text{ are integers}
   \]
   \[
   \text{THEN } J(s) \ast J(t)
   \]
   \[
   \text{ELSE UNDEF}
   \]

\( J(s/t) = \)
   \[
   \text{IF } J(s) \text{ and } J(t) \text{ are integers and}
   \]
   \[
   J(s) \text{ is not 0}
   \]
   \[
   \text{THEN } J(s) / J(t) \text{ (integer divide)}
   \]
   \[
   \text{ELSE UNDEF}
   \]

\( J(s^{**}t) = \)
   \[
   \text{IF } J(s) \text{ is an integer and}
   \]
   \[
   J(t) \text{ is an integer not less than 0}
   \]
   \[
   \text{THEN } J(s) J(t)
   \]
   \[
   \text{ELSE UNDEF}
   \]

\( J(t[i]) = \)
   \[
   \text{IF } J(t) \text{ is an array and}
   \]
   \[
   J(i) \text{ is an integer and}
   \]
   \[
   J(t)_{lb} \leq J(i) \leq J(t)_{ub}
   \]
   \[
   \text{THEN } \{J(t)\}_{J(i)-J(t)_{lb}+1}
   \]
   \[
   \text{ELSE UNDEF}
   \]

\( J(t[i:j]) = \)
   \[
   \text{IF } J(t) \text{ is an array and}
   \]
   \[
   J(i) \text{ and } J(j) \text{ are integers and}
   \]
   \[
   J(t)_{lb} \leq J(i) \leq J(j) \leq J(t)_{ub}
   \]
   \[
   \text{THEN } < J(i), J(j),
   \]
   \[
   < J(t[i]), ..., J(t[j]) >
   \]
   \[
   \text{ELSE UNDEF}
   \]

\( J(t[i<-s]) = \)
   \[
   \text{IF } J(t) \text{ is an array and}
   \]
   \[
   J(i) \text{ is an integer and}
   \]
   \[
   J(s) \text{ is an integer}
   \]
   \[
   \text{THEN IF } J(t)_{lb} \leq J(i) \leq J(t)_{ub}
   \]
   \[
   \text{THEN } < J(t)_{lb}, J(t)_{ub},
   \]
   \[
   [J(t)](J(i)-J(t)_{lb}+1 \leftarrow J(s)) >
   \]
   \[
   \text{ELSE } J(t)
   \]
   \[
   \text{ELSE UNDEF}
   \]
\( J(g(t_1, \ldots, t_n)) = \)
\begin{align*}
&\text{IF none of } J(t_1), \ldots, J(t_n) \text{ are UNDEF} \\
&\text{THEN the result of applying the n-ary function} \\
&\text{named } g \text{ to arguments } J(t_1), \ldots, J(t_n) \\
&\text{ELSE UNDEF}
\end{align*}

\( J(\text{if } P \text{ then } t \text{ else } s) = \)
\begin{align*}
&\text{IF } J(P) \text{ is T} \\
&\text{THEN } J(t) \\
&\text{ELSE } J(s)
\end{align*}

The remaining cases of the definition of \( J \) map formulas into truth or falsehood.

\( J(\text{TRUE}) = T \)
\( J(\text{FALSE}) = F \)

\( J(\text{if } P \text{ then } Q \text{ else } R) = \)
\begin{align*}
&\text{IF } J(P) \text{ is T} \\
&\text{THEN } J(Q) \\
&\text{ELSE } J(R)
\end{align*}

\( J(s=t) = \)
\begin{align*}
&\text{IF } J(s) \text{ and } J(t) \text{ are integers} \\
&\text{THEN IF } J(s) = J(t) \\
&\text{THEN } T \\
&\text{ELSE } F \\
&\text{ELSE IF } J(s) \text{ and } J(t) \text{ are arrays and} \\
&\text{J(s)}_{\text{ub}} = J(t)_{\text{ub}} \text{ and} \\
&\text{J(s)}_{\text{lb}} = J(t)_{\text{lb}} \text{ and} \\
&\text{J(s[i]) = J(t[i])} \\
&\text{for all } i \text{ between } J(s)_{\text{lb}} \text{ and } J(s)_{\text{ub}} \\
&\text{THEN } T \\
&\text{ELSE } F
\end{align*}

\( J(s \leq t) = \)
\begin{align*}
&\text{IF } J(s) \text{ and } J(t) \text{ are integers} \\
&\text{THEN IF } J(s) \leq J(t) \\
&\text{THEN } T \\
&\text{ELSE } F \\
&\text{ELSE } F
\end{align*}
\[ J(p(t_1, \ldots, t_n)) = \]
\begin{align*}
&\text{IF none of } J(t_1), \ldots, J(t_n) \text{ are UNDEF} \\
&\text{THEN the result of applying the } n\text{-ary predicate} \\
&\text{named } p \text{ to arguments } J(t_1), \ldots, J(t_n) \\
&\text{ELSE } F
\end{align*}

Proof Theory

Notation and Terminology

If \( P, Q, \) and \( R \) are formulas, then \( P[Q \text{ for } R] \) is the formula obtained from \( P \) by replacing all occurrences of \( R \) by \( Q \).

Rewrite rules are of the form:

\[ R \implies Q \]

where \( R \) and \( Q \) are formula schemas. Applying such a rewrite rule to a formula \( P \) consists of replacing all occurrences of \( R \) in \( P \) by \( Q \); thus, the rewritten formula is \( P[Q \text{ for } R] \).

Proofs

A proof of a formula \( P \) in AL is a sequence of formulas, where the first and last formulas in the sequence are \( P \) and \( \text{TRUE} \), respectively, and each formula in the sequence is obtained from the preceding one by applying one of the rewrite rules given below or a rewrite rule derived from a user-supplied axiom (see Appendix E). For the subset of AL that corresponds to the propositional calculus, the rewrite rules given below are complete, i.e., every true formula is provable and vice-versa. The proof of completeness is essentially the one in Guttag et al. (1976).
Rewrite Rules

In the rewrite rules below, the following conventions are used:

\[ P, P_1, P_2, Q, R \] are \textit{formula}s
\[ s, s_1, s_2, t, t_1, t_2 \] are \textit{term}s

1. Definitional rules for if.

\[
\begin{align*}
\text{(if TRUE then } P \text{ else } Q) & \Rightarrow P \\
\text{(if FALSE then } P \text{ else } Q) & \Rightarrow Q \\
\text{(if TRUE then } s \text{ else } t) & \Rightarrow s \\
\text{(if FALSE then } s \text{ else } t) & \Rightarrow t
\end{align*}
\]

2. Repeated result rules.

\[
\begin{align*}
\text{(if } P \text{ then } Q \text{ else } Q) & \Rightarrow Q \\
\text{(if } P \text{ then } t \text{ else } t) & \Rightarrow t
\end{align*}
\]

3. Redundant if rule.

\[
\text{(if } P \text{ then TRUE else FALSE) } \Rightarrow P
\]

4. If distribution rules.

\[
\begin{align*}
\text{(if (if } P \text{ then } Q \text{ else } R) \text{ then } P_1 \text{ else } P_2) & \Rightarrow \\
\text{(if } P \text{ then (if } Q \text{ then } P_1 \text{ else } P_2) \text{ else (if } R \text{ then } P_1 \text{ else } P_2) & \\
\text{(if (if } P \text{ then } Q \text{ else } R) \text{ then } s \text{ else } t) & \Rightarrow \\
\text{(if } P \text{ then (if } Q \text{ then } s \text{ else } t) \text{ else (if } R \text{ then } s \text{ else } t) &
\end{align*}
\]

5. Logical substitution rules.

\[
\begin{align*}
\text{(if } P \text{ then } Q \text{ else } R) & \Rightarrow \\
\text{(if } P \text{ then } Q[\text{TRUE for } P] \text{ else } R[\text{FALSE for } P]) & \\
\text{(if } P \text{ then } s \text{ else } t) & \Rightarrow \\
\text{(if } P \text{ then } s[\text{TRUE for } P] \text{ else } t[\text{FALSE for } P]) &
\end{align*}
\]

\[ s = s \implies \text{TRUE} \]

\[(\text{if } s = t \text{ then } P \text{ else } Q) \implies (\text{if } s = t \text{ then } P s \text{ for } t \text{ else } Q \text{[FALSE for } t=s]) \]

7. Rules for inequality.

\[ s < s \implies \text{TRUE} \]

\[(\text{if } s < t \text{ then } P \text{ else } Q) \implies (\text{if } s < t \text{ then } P s=t \text{ for } t<s \text{ else } Q \text{[TRUE for } t<s]) \]

\[(\text{if } s < t \text{ then } P \text{ else } Q) \implies (\text{if } s < t \text{ then } P \text{[FALSE for } s=t] \text{ else } Q \text{[FALSE for } t=s])\]

8. Restricted case analysis rules. The rules below are given for the equality operator =, but apply to the inequality operator \(<\) as well.

\[(\text{if } P \text{ then } s_1 \text{ else } s_2) = t \implies (\text{if } P \text{ then } s_1=t \text{ else } s_2=t) \]

\[ s = (\text{if } P \text{ then } t_1 \text{ else } t_2) \implies (\text{if } P \text{ then } s=t_1 \text{ else } s=t_2) \]
APPENDIX C

PROOF OF THE WEAKEST PRECONDITION FUNCTION FOR L

Derived Rules of Inference for L

In order for a semantic definition of a programming language to be satisfactory, it must satisfy the axioms set forth by Floyd (1967). For an axiomatic description, this means that the semantics must include some special rules of inference that "implement" these axioms. Two of these rules are especially important here, since they are needed in order to prove that the weakest precondition function given earlier for L does in fact compute the weakest precondition. These two rules are:

DR1:

\[
\begin{align*}
& P \rightarrow P', \quad \{P'\} \ A \ \{Q\} \\
\hline
& \{P\} \ A \ \{Q\}
\end{align*}
\]

DR2:

\[
\begin{align*}
& \{P\} \ A \ \{Q'\}, \ Q' \rightarrow Q \\
\hline
& \{P\} \ A \ \{Q\}
\end{align*}
\]

where \(P, P', Q,\) and \(Q'\) are formulas, and \(A\) is a list of one or more statements in L. The reader will perhaps have noticed that these rules have not been explicitly included in the semantic description of L given in Appendix A. This is because they, and the other rules of inference needed to
satisfy Floyd's axioms, are derivable from the primitive rules of inference.

**Theorem 1.** DR1 and DR2 are derived rules of inference for L.

**Proof.** Since the proofs for DR1 and DR2 are so similar, only the proof for DR1 is given here.

Let \( P, P', \) and \( Q \) be formulas and \( A \) be a list of one or more L statements. The theorem is proved for DR1 if the following can be shown: if \( P \rightarrow P' \) is true and if \( \{P'\} A \{Q\} \) is provable, then \( \{P\} A \{Q\} \) is provable.

The proof is by induction on the number of statements in \( A \). Since the proof would be quite lengthy if presented in full, only representative cases of the basis and inductive steps are presented.

**(Basis)** \( A \) is a single statement with no imbedded statements. There are four cases to consider, corresponding to the assignment, let, null, and lemma statements.

(i) (a) \( A \) is of the form "\( x:=fx \)." Assume \( P \rightarrow P' \) and \( \{P'\}x:=fx\{Q\} \). If \( \{P'\}x:=fx\{Q\} \) is provable, it is provable using only the primitive rules of inference. Thus, since the only primitive rule of inference available to prove \( \{P'\}x:=fx\{Q\} \) is PR3 (see Appendix A), \( P' \rightarrow Q(x/fx) \) must be true. Since \( P \rightarrow P' \), \( P \rightarrow Q(x/fx) \). Applying rule PR3 yields \( \{P\}x:=fx\{Q\} \).

(b) \( A \) is of the form "\( x[i]:=fx \)." Similar to (a).
(ii) -- (iv) Proof is similar to (i) above.

(Induction) A is composed of more than one statement, either through statement imbedding or because A is a list of statements. There are five cases, corresponding to a statement list, the compound, if, and while statements, and a program consisting of an entry statement followed by a statement list.

(i) A is of the form "A';S", where A' is a list of one or more L statements and S is a statement. Assume P->P' and {P}'A';S{Q}. If {P}'A';S{Q} is provable, it is provable using only the primitive rules of inference. Thus, since PR2 is the only primitive rule of inference available to prove {P}'A';S{Q}, there is some R for which {P}'A'{R} and {R}S{Q}. By induction, {P}A'{R} can be concluded from P->P' and {P}'A'{R}. Applying rule PR2 yields {P}A';S{Q}.

(ii) -- (v) Proof is similar to (i) above. ☐

The Proof

In order to show that the function [A]Q defined for L actually computes the weakest precondition as intended, it suffices to show, where P and Q are formulas and A is a list of one or more statements of L:

1. [A]Q is a precondition, i.e., {[A]Q}A{Q}.  

2. \( [A]Q \) is the weakest precondition, i.e., if \( \{P\}A\{Q\} \) then \( P \rightarrow [A]Q \).

**Theorem 2.** Let \( A \) be a list of one or more statements in \( L \), and \( P \) and \( Q \) be formulas. Then \( \{[A]Q\}A\{Q\} \).

**Proof.** Since \( [A]Q \rightarrow [A]Q \) is a tautology, it suffices to show that, for any formula \( P \), if \( P \rightarrow [A]Q \) then \( \{P\}A\{Q\} \). The proof is by induction on the number of statements in \( A \).

**(Basis)** \( A \) is a single statement. There are four cases to consider.

(i) (a) \( A \) is of the form \( "x:=fx" \). Assume \( P \rightarrow [x:=fx]Q \), i.e., \( P \rightarrow Q(x/fx) \). By rule PR3, \( \{P\}x:=fx\{Q\} \).

(b) \( A \) is of the form \( "x[i]:=fx" \). Similar to (a).

(ii) \( A \) is of the form \( "let x:=fx" \). Similar to (i) (a) above.

(iii) \( A \) is \( "null" \). Assume \( P \rightarrow [null]Q \), i.e., \( P \rightarrow Q \). By rule PR8, \( \{P\}null\{Q\} \).

(iv) \( A \) is of the form \( "lemma T assert R" \). Assume \( P \rightarrow [lemma T assert R]Q \), i.e., \( P \rightarrow T \rightarrow R \& Q \). Since this formula is equivalent to \( P \& T \rightarrow R \& Q \), rule PR9 may be applied to yield \( \{P\}lemma T assert R\{Q\} \).

**(Induction)** \( A \) is composed of more than one statement,
either through statement imbedding, or because A is a list of statements. There are five cases.

(i) A is of the form "A';S", where A' is a list of one or more L statements and S is a statement. Assume P->[A';S]Q, i.e., P->[A'](S]Q. By the induction hypothesis, {P}A'{[S]Q} and, since [S]Q->[S]Q, {[S]Q}S{Q}. Applying rule PR2 with R=[S]Q yields {P}A';S{Q}.

(ii) A is of the form "if B then S1 else S2". Assume P->[if B then S1 else S2]Q, i.e.,

P->(if B then [S1]Q else [S2]Q)

from which the two formulas P&B->[S1]Q and P&-B->[S2]Q may be inferred. By the inductive hypothesis, {P&B}S1{Q} and {P&-B}S2{Q}, so that, by rule PR7,

{P}if B then S1 else S2{Q}

(iii) A is of the form "lemma T assert R while B do S". Assume P->[lemma T assert R while B do S]Q. If P is false, then {P}A{Q} is vacuously true. Otherwise, by the definition of [A]Q for the while statement, in order for P->[A]Q to be true, [lemma T assert R while B do S]Q equals R, which can only be the case if T&R&B->[S]R and T&R&-B->[Q] are both true. By the induction hypothesis, {T&R&B}S{R}. Thus, the three conditions needed to apply rule PR10 are met, yielding {P}lemma T assert R while B do S{Q}.
(iv) A is of the form "begin S1; S2; ...; Sn end". Assume P->[begin S1; S2; ...; Sn end], i.e., P->[S1; S2; ...; Sn]Q. By the induction hypothesis, {P}S1; S2; ...; Sn{Q}. Applying rule PR6 yields {P}begin S1; S2; ...; Sn end{Q}.

(v) A is of the form "entry R; A'", where A' is a list of one or more statements. Assume P->[entry R; A']Q. If P is false, then {P}A{Q} is vacuously true. Otherwise, by the definition of [A]Q, in order for P->[A]Q to be true, R->[A']Q must be true. By the induction hypothesis, {R}A'{Q}. Applying rule PR1 yields {P}entry R; A'{Q}. □

Theorem 3. Let A be a list of one or more statements in L, and P and Q be formulas. If {P}A{Q} then P->[A]Q.

Proof. The proof is by induction on the number of statements in A. Since the proof is straightforward but lengthy, only representative cases of the basis and inductive steps are presented below.

(Basis) A is a single statement. There are four cases to consider, corresponding to the assignment, let, null, and lemma statements.

(i) (a) A is of the form "x:=fx". Assume {P}x:=fx{Q}. Since PR3 is the only primitive rule of inference from which
{P}x:=fx{Q} can be proved, P\rightarrow Q(x/fx) must be true. But Q(x/fx) is \([x:=fx]Q\), and thus, P\rightarrow[x:=fx]Q.

(b) A is of the form "x[i]:=fx". Similar to (a).

(ii) -- (iv) Proof is similar to (i) above.

(Induction) A is composed of more than one statement, either through statement imbedding, or because A is a list of statements. There are five cases, corresponding to a statement list, the compound, if, and while statements, and a program consisting of an entry statement followed by a statement list.

(i) A is of the form "A';S", where A' is a list of one or more statements and S is a statement. Assume \{P\}A';S{Q}. Since PR2 is the only primitive rule of inference from which \{P\}A';S{Q} can be proved, there is some R for which \{P\}A'{R} and \{R\}S{Q}. By induction, R\rightarrow[S]Q, and by rule DR2, \{P\}A'\{[S]Q\}. Applying induction once again yields P\rightarrow[A'][S]Q, i.e., P\rightarrow[A';S]Q.

(ii) -- (v) Proof is similar to (i) above. □

Theorem 4. There is an effective way of computing the weakest precondition function for L.

Proof. The theorem follows directly from the preceding two theorems and the definition of the weakest precondition function for L. □
APPENDIX D

INCREMENTAL VERIFICATION ALGORITHM FOR L

Chief Executive

\[ \text{entry R;A} \]

\begin{align*}
&\text{BEGIN} \\
&\quad \text{hire a subordinate \text{SUB}_A to find \{A\}TRUE;} \\
&\quad \text{FIRST precondition \text{Q}' received from \text{SUB}_A} \\
&\quad \text{\text{SUCHTHAT} R->Q' is provable} \\
&\quad \text{DO BEGIN} \\
&\quad \quad \text{tell \text{SUB}_A that Q' is satisfactory;} \\
&\quad \quad \text{succeed} \\
&\quad \quad \text{END} \\
&\quad \text{OTHERWISE fail} \\
&\text{END}
\end{align*}

Subordinates

\[ [x:=fx]Q \quad \text{and} \quad \text{let } x:=fx]Q \]

\begin{align*}
&\text{BEGIN} \\
&\quad \text{replace all occurrences of x in Q by fx;} \\
&\quad \text{succeed, returning Q;} \\
&\quad \text{IF my superior finds Q satisfactory} \\
&\quad \text{THEN do nothing} \\
&\quad \text{ELSE fail} \\
&\text{END}
\end{align*}
\[ x[i] := fx \] Q

\begin{verbatim}
BEGIN
  for all j, replace all occurrences of x[j] in Q
  by (if i=j then fx else x[j]) and, simultaneously,
  replace all unsubscripted occurrences of x in Q
  by x[i<-fx];
  succeed, returning Q;
  IF my superior finds Q satisfactory
  THEN do nothing
  ELSE fail
END

[A;S]Q

BEGIN
  hire a subordinate SUB_S to find [S]Q;
  WHILE SUB_S returns a precondition Q' DO
    BEGIN
      hire a subordinate SUB_A to find [A]Q';
      WHILE SUB_A returns a precondition Q'' DO
        BEGIN
          succeed, returning Q'';
          IF my superior finds Q'' satisfactory
          THEN BEGIN
            tell SUB_S that Q' is satisfactory;
            tell SUB_A that Q'' is satisfactory
          END
        END
      END
    END
  END;
  fail
END
\end{verbatim}
\[ \text{begin A end} Q \]

BEGIN
hire a subordinate \( \text{SUB}_A \) to find [A]Q;

WHILE \( \text{SUB}_A \) returns a precondition Q' DO
BEGIN
succeed, returning Q';
IF my superior finds Q' satisfactory
THEN tell \( \text{SUB}_A \) that Q' is satisfactory
END;
fail
END

\[ \text{if B then S1 else S2} Q \]

BEGIN
hire a subordinate \( \text{SUB}_{S1} \) to find [S1]Q;

WHILE \( \text{SUB}_{S1} \) returns a precondition Q1 DO
BEGIN
hire a subordinate \( \text{SUB}_{S2} \) to find [S2]Q;

WHILE \( \text{SUB}_{S2} \) returns a precondition Q2 DO
BEGIN
let Q' be (if B then Q1 else Q2);
succeed, returning Q';
IF my superior finds Q' satisfactory
THEN BEGIN
tell \( \text{SUB}_{S1} \) that Q1 is satisfactory;
tell \( \text{SUB}_{S2} \) that Q2 is satisfactory
END
END
END;
fail
END

\[ \text{lemma R1 assert R2} Q \]

BEGIN
let Q' be R1->R2&Q;
succeed, returning Q';
IF my superior finds Q' satisfactory
THEN replace the program segment by "\text{lemma R1&R2}"
ELSE fail
END
BEGIN
set up a generator G of alternate invariants that subsume the initial asserted invariant I;

REPEAT
IF J&I&¬B -> Q is provable THEN
BEGIN
hire a subordinate SUBS to find [S]I;
FIRST precondition I' returned by SUBS
SUCHTHAT ISINDUCTIVE(I)
DO BEGIN
succeed, returning I;
IF my superior is satisfied with I THEN
BEGIN
conjoin I to the formula on the lemma clause;
erase the assert clause;
tell SUBS that I' is satisfactory
END
END
OTHERWISE do nothing
END;
request an alternate invariant I from G
UNTIL G offers no more alternate invariants;
fail
END

ISINDUCTIVE(I):

IF J&I&B -> I'
THEN I is inductive
ELSE BEGIN
set up a generator H of alternate invariants;
FIRST formula K returned by H
SUCHTHAT J&I&B&K -> I' and
ISINVARINT(K)
DO I is inductive
OTHERWISE I is not inductive
END
ISINVARIANT(K):

BEGIN
erase all assert clauses from the entire program P;
add "assert K" to this while statement;
hire another "chief executive" to verify the new program;
IF this chief executive succeeds
THEN K is invariant
ELSE K is not invariant;
restore the erased assert clauses
END

[null]Q

BEGIN
succeed, returning Q;
IF my superior is satisfied with Q
THEN do nothing
ELSE fail
END
One of the ways in which the user interacts with IFS is through the axiom command. From the user's point of view, each axiom command specifies that a certain formula, derived from the command, is to be assumed true. IFS treatment of the command is to apply a certain rewrite rule, also derived from the command, to the formula being simplified at the time. This appendix shows that the rewrite rule used by IFS for a given axiom command is justified with respect to the formula which is to be assumed true.

The six forms of the axiom command are given below, each with its derived formula and rewrite rule. A justification is given for the rewrite rules with respect to their associated formulas. Each justification is as follows. Let L be the formula derived from an axiom command and R the rewrite rule. Let A' be the formula obtained from an arbitrary formula A by applying R. R is justified with respect to L if, assuming L is true, it can be shown that if A' is true, then A is true.
Notation

Let A, P, and Q be formulas, s and t terms.

A[Q for P] is the formula obtained from A by replacing all occurrences of P by Q. Similarly, A[s for t] is the formula obtained from A by replacing all occurrences of t by s.

A[[Q for P]] is the formula obtained from A by replacing all occurrences of P in the consequent of A by Q. Thus, A[[Q for P]] is defined (recursively) as follows, where B, C, and D are arbitrary formulas:

\[
A[[Q for P]] =
\]

IF A is P
THEN Q
ELSE IF A is (if B then C else D)
THEN (if B then C[[Q for P]] else D[[Q for P]])
ELSE IF A is (B -> C)
THEN (B -> C[[Q for P]])
ELSE IF A is (B or C)
THEN (B[[Q for P]] or C[[Q for P]])
ELSE IF A is (B & C)
THEN (B[[Q for P]] & C[[Q for P]])
ELSE A

Justifications

axiom s is t

formula: s=t
rewrite rule: s => t

Theorem 1. Assuming s=t is true, if A[t for s] is true, then A is true.

Proof. Assume s=t and A[t for s] are true. Note
that \( A[t \text{ for } s] \) remains true if equals are substituted for equals. Thus, substitute \( s \) for each occurrence of \( t \) that is a replacement for \( s \) in \( A[t \text{ for } s] \). \( A \) is the resulting formula. 

\[ \text{axiom if } P \text{ then } s \text{ is } t \]

\[ \text{formula: } \ P \rightarrow s=t \]
\[ \text{rewrite rule: } \ s \Rightarrow (\text{if } P \text{ then } t \text{ else } s) \]

**Theorem 2.** Let \( A' \) be \( A[(\text{if } P \text{ then } t \text{ else } s) \text{ for } s] \). Assuming \( P \rightarrow s=t \) is true, if \( A' \) is true, then \( A \) is true.

\[ \text{Proof. } \text{Assume } P \rightarrow s=t \text{ and } A' \text{ are true. If } P \text{ is true, then } s=t \text{ is true and } A' \text{ reduces to } A[t \text{ for } s]. \]
\( \text{Theorem 1 applies, so that } A \text{ is true. Otherwise (i.e., if } P \text{ is not true) then } A' \text{ reduces to } A[s \text{ for } s], \text{ that is, } A. \]

\[ \text{axiom } P \text{ iff } Q \]
\[ \text{formula: } \ P \leftrightarrow Q \]
\[ \text{rewrite rule: } P \Rightarrow Q \]

**Theorem 3.** Assuming \( P \leftrightarrow Q \) is true, if \( A[Q \text{ for } P] \) is true, then \( A \) is true.

\[ \text{Proof. } \text{Assume } P \leftrightarrow Q \text{ and } A[Q \text{ for } P] \text{ are true. Since } Q \text{ is true exactly when } P \text{ is true, } A[Q \text{ for } P] \text{ remains true if occurrences of } Q \text{ are replaced by } P. \text{ Thus, replace } P \text{ for each occurrence of } Q \text{ that is a replacement for } P \text{ in } A[Q \text{ for } P]. \text{ } A \text{ is the resulting formula}. \]
axiom if R then P iff Q

formula: \( R \rightarrow (P \leftrightarrow Q) \)
rewrite rule: \( P \Rightarrow (if \ R \ then \ Q \ else \ P) \)

Theorem 4. Let \( A' \) be \( A[(if \ R \ then \ Q \ else \ P) \ for \ P] \). Assuming \( R \rightarrow (P \leftrightarrow Q) \) is true, if \( A' \) is true, then \( A \) is true.

Proof. Assume \( R \rightarrow (P \leftrightarrow Q) \) and \( A' \) are true. If \( R \) is true, then \( P \leftrightarrow Q \) is true and \( A' \) reduces to \( A[Q \ for \ P] \). Theorem 3 applies, so that \( A \) is true. Otherwise (i.e., if \( R \) is not true) then \( A' \) reduces to \( A[P \ for \ P] \), that is, \( A \). □

axiom P if Q

formula: \( Q \rightarrow P \)
rewrite rule: \( P \Rightarrow Q \) (to be applied only in the consequent of a formula)

Theorem 5. Assuming \( Q \rightarrow P \) is true, if \( A[[Q \ for \ P]] \) is true, then \( A \) is true.

Proof. Assume \( Q \rightarrow P \) and \( A[[Q \ for \ P]] \) is true. The proof is by induction on \( A \).

(Basis) \( A \) is \( P \). Then \( A[[Q \ for \ P]] \) is \( Q \). Since both \( Q \rightarrow P \) and \( Q \) are true, \( P \) is also true. That is, \( A \) is true.

(Induction) There are five cases, corresponding to the last five cases of the definition of \( A[[Q \ for \ P]] \). The proof of only one case is given here, the proofs for the remaining cases being either similar or trivial.
Suppose $A$ is $(\text{if } B \text{ then } C \text{ else } D)$ for some formulas $B$, $C$, and $D$. Then $A[\{Q \text{ for } P\}]$ is $(\text{if } B \text{ then } C[\{Q \text{ for } P\}]$ \\
else $D[\{Q \text{ for } P\}]$). If $B$ is true, then $A[\{Q \text{ for } P\}]$ reduces \\
to $C[\{Q \text{ for } P\}]$. Thus, $C[\{Q \text{ for } P\}]$ is true, and by induc-

tion, $C$ is true. Since if $B$ is true, $A$ also reduces to $C$, $A$ \\
is true. Otherwise (i.e., if $B$ is not true) then $A[\{Q \text{ for } P\}]$ reduces to \\
$D[\{Q \text{ for } P\}]$. Thus, $D[\{Q \text{ for } P\}]$ is true, \\
and by induction, $D$ is true. Since if $B$ is not true, $A$ also \\
reduces to $D$, $A$ is true. □

axiom if $R$ then $P$ if $Q$

formula: $R \rightarrow (Q \rightarrow P)$

rewrite rule: $P \Rightarrow (\text{if } R \text{ then } Q \text{ else } P)$

(to be applied only in the 
consequent of a formula)

**Theorem 6.** Let $A'$ be $A[\{\text{if } R \text{ then } Q \text{ else } P\} \text{ for } P]$. 
Assuming $R \rightarrow (Q \rightarrow P)$, if $A'$ is true, then $A$ is true.

**Proof.** Assume $R \rightarrow (Q \rightarrow P)$ and $A'$ are true. If $R$ 
is true, then $Q \rightarrow P$ is true and $A'$ reduces to $A[Q \text{ for } P]$. 
Theorem 3 applies, so that $A$ is true. Otherwise (i.e., if $R$ 
is not true) then $A'$ reduces to $A[P \text{ for } P]$, that is, $A$. □
APPENDIX F

EXAMPLE PROGRAMS AND TRANSCRIPTS

The prototype incremental verification system has been applied to a number of example programs. Three of the most substantial of these appear in this appendix, together with some transcripts produced by the prototype verification system.

In the transcripts below the user has typed everything in a line that occurs after a # sign and the following prompts:

  input:
  trial invariant:
  suggest an R:

When the last character of a line typed by the user is a hyphen, the user's response continues onto the next line.

Due to character set restrictions, the prototype verification system recognizes the following symbol combinations, which appear in the transcripts below.

\ = for \neq
\leq = for \leq
\geq = for \geq
User-Defined Predicates and Functions

The transcripts below use the following user-defined predicates and functions with the indicated meanings:

\[
\text{exch}(x,y,z) =
\begin{align*}
&\text{IF } x \text{ is an array and } \\
&\text{y and } z \text{ are integers} \\
&\text{THEN IF } x_{lb} \leq y < x_{ub} \text{ and } \\
&\quad x_{lb} \leq z < x_{ub} \\
&\quad \text{THEN } x[y<-x[z]][z<-x[y]] \\
&\quad \text{ELSE } x \\
&\quad \text{ELSE undefined}
\end{align*}
\]

\[
\text{isin}(x,y) =
\begin{align*}
&\text{IF } x \text{ is an integer and } \\
&\text{y is an array} \\
&\text{THEN IF } x \text{ occurs as the value of one of } \\
&\quad \text{the elements of } y \\
&\quad \text{THEN true} \\
&\quad \text{ELSE false} \\
&\quad \text{ELSE false}
\end{align*}
\]

\[
\text{issorted}(x) =
\begin{align*}
&\text{IF } x \text{ is an array} \\
&\text{THEN IF the elements of } x \text{ are in ascending order} \\
&\quad \text{THEN true} \\
&\quad \text{ELSE false} \\
&\quad \text{ELSE false}
\end{align*}
\]

\[
\text{part}(x,y) =
\begin{align*}
&\text{IF } x \text{ is an array and } \\
&\text{y is an integer} \\
&\text{THEN IF each element of } x[y_{lb}:y] \text{ is} \\
&\quad \text{less than or equal to each element in } x[(y+1):x_{ub}] \\
&\quad \text{THEN true} \\
&\quad \text{ELSE false} \\
&\quad \text{ELSE false}
\end{align*}
\]

\[
\text{least}(x) =
\begin{align*}
&\text{IF } x \text{ is an array} \\
&\text{THEN } i \text{ such that } x[i] \text{ is the smallest element} \\
&\quad \text{in } x \\
&\quad \text{ELSE undefined}
\end{align*}
\]
Binary Search Program

The binary search example was first discussed in Chapter 3 and was mentioned again in Chapter 5. The transcript produced as a result of interacting with the prototype verification system to verify this program follows.

input: srch1

entry (b <= c &
   issorted(a[b:c]));
let b0:= b;
let c0:= c;
while(loop) b \= c do
begin
   d:=((b + c) / 2);
   if key > a[d] then
      b:=(d + 1)
   else
      c:= d
end;
assert (isin(key, a[b0:c0])
   ->
   key = a[b])

((~ b \= c &
   (TRUE &
    TRUE))
   ->
   (isin(key, a[b0:c0])
    ->
    key = a[b]))
#

while(loop) b \= c do ...
assert (isin(key, a[b0:c0])
   ->
   key = a[b])

trial invariant: (isin(key,a[b0:c0])-
   -> isin(key,a[b:c])

((isin(key, a[b0:c0])
   ->
   isin(key, a[b:c]))
   ->
   TRUE)
# reduce
true

#

((\neg b \neq c &
  (true &
   (isin(key, a[b0:c0])
    ->
     isin(key, a[b:c])))
  ->
   (isin(key, a[b0:c0])
    ->
     key = a[b])))

# reduce


# axiom key=a[b] iff isin(key,a[b:b])

# reduce

true

#

((\neg b \neq c &
  (true &
   (isin(key, a[b0:c0])
    ->
     isin(key, a[b:c])))
  ->
   (if key > a[((b + c) / 2)]
    then(isin(key, a[b0:c0])
     ->
      isin(key, a[((b + c) / 2) + 1]:c))
    else(isin(key, a[b0:c0])
     ->
      isin(key, a[b:((b + c) / 2)]))))

# s=t z=((b+c)/2)

# reduce

((z =((b + c) / 2) &
  (b \neq c &
   (isin(key, a[b0:c0]) &
    isin(key, a[b:c])))
  ->
   (if key <= a[z]
    then isin(key, a[b;z])
    else isin(key, a[(z + 1):c])))

# axiom if isin(key,a[b:c]) then-

(if key<=a[z]-
  then isin(key,a[b;z])-
  else isin(key,a[(z+1):c])) if (issent@a[b:c]) &-
  (b<=z & z<=c))
# reduce

\[
\frac{z = ((b + c) / 2) \land (b \not= c \land (\text{isin(key, a[b0:c0]) \land \text{isin(key, a[b:c]]))))}{\text{issorted(a[b:c]) \land (b \leq z \land z \leq c))}}
\]

#

while (loop) b \not= c do ...

The assertion \((\text{isin(key, a[b0:c0])} \Rightarrow \text{isin(key, a[b:c])})\)

is invariant if R is, where

\[
R \Rightarrow ((z = ((b + c) / 2) \land (b \not= c \land (\text{isin(key, a[b0:c0]) \land \text{isin(key, a[b:c]])))) \Rightarrow (\text{issorted(a[b:c]) \land (b \leq z \land z \leq c)})))
\]

suggest an R: \((\text{issorted(a[b:c]) \land b \leq c})\)

\[
((\text{issorted(a[b:c]) \land b \leq c}) \Rightarrow ((z = ((b + c) / 2) \land (b \not= c \land (\text{isin(key, a[b0:c0]) \land \text{isin(key, a[b:c]])))) \Rightarrow (\text{issorted(a[b:c]) \land (b \leq z \land z \leq c)})))
\]

# axiom if z=((b+c)/2) then-

\[
(b \leq z \land z \leq c) \text{ if } b \leq c
\]

# reduce

TRUE

#

\[
((\neg b \not= c \land (\text{TRUE} \land (\text{issorted(a[b:c]) \land b \leq c}))) \Rightarrow \text{TRUE})
\]
# reduce
TRUE
#

((b \not= c &
TRUE &
(issorted(a[b:c]) &
b <= c)))
->
(if key > a[((b + c) / 2)]
then(issorted(a[((b + c) / 2) + 1):c]) &

(((b + c) / 2) + 1) <= c)
else(issorted(a[b:((b + c) / 2)]) &
b <=((b + c) / 2))))

# set z=((b+c)/2)
# display
(z =((b + c) / 2)
->
((b \not= c &
TRUE &
(issorted(a[b:c]) &
b <= c)))
->
(if key > a[z]
then(issorted(a[(z + 1):c]) &
(z + 1) <= c)
else(issorted(a[b:z]) &
b <= z)))

# axiom if issorted(a[b:c]) then-
issorted(a[(z+1):c]) if (b<=(z+1) & (z+1)<=c)
# axiom if issorted(a[b:c]) then-
issorted(a[b:z]) if (b<=z & z<=c)

# reduce
((z =((b + c) / 2) &
(b \not= c &
(issorted(a[b:c]) &
b <= c)))
->
(if key <= a[z]
then(b <= z &
z <= c)
else(b <=(z + 1) &
(z + 1) <= c)))
# axiom b<=(z+1) if b<=z

# axiom z<=c if (z+1)<=c
# reduce
((z =((b + c) / 2) &
(b \not= c &
(issorted(a[b:c]) &
b <= c)))
->
(b <= z &
        (z + 1) <= c))
# axiom if z=((b+c)/2) then-
    b<=z if b<=c
# axiom if z=((b+c)/2) then-
    (z+1)<=c if b<c
# reduce
    TRUE
#

((b <= c &
    issorted(a[b:c]))
  ->
    (issorted(a[b:c]) &
        b <= c))
# reduce
    TRUE
#

((b <= c &
    issorted(a[b:c]))
  ->
    (isin(key, a[b:c])
      ->
        isin(key, a[b:c])))
# reduce
    TRUE
#
Assertions verified.
output: srch2

The program written to file srch2 is:

entry (b <= c &
    issorted(a[b:c]));
let b0:= b;
let c0:= c;
lemma ((issorted(a[b:c]) &
            b <= c) &
            (isin(key, a[b0:c0]))
      ->
        isin(key, a[b:c]))
while(loop) b \= c do
begin
  d:=((b + c) / 2);
  if key > a[d] then
    b:=(d + l)
  else
    c:= d
end;
lemma (isin(key, a[b0:c0])
    ->
    key = a[b])

Selection Sort Program

The following program performs a selection sort for an array a[l:n].

entry 1<=n;
i:=0;
while(outer) i<n do
begin
    i:=(i+1);
j:=i;
x:=a[i];
k:=i;
lemma j<=n
while(inner) j<n do
begin
    j:=(j+1);
    if a[j]<x
        then begin x:=a[j]; k:=j end
        else null
end;
a[k]:=a[i];
a[i]:=x
end

The first pass through the incremental verifier validates the assertion j=n at termination of the inner loop and incidentally contributes j<n as a verified invariant for the inner loop. Since this pass is not very interesting, its transcript is omitted here.

The second pass verifies that upon termination of the inner loop, a[k] is the least element of a[i:n] and x equals a[k]. The transcript for this pass follows.

input: selectl

entry 1 <= n;
i:= 0;
while(outer)  i < n do
begin
  i:=(i + 1);
  j:= i;
  x:= a[i];
  k:= i;
lemma  j <= n
while(inner)  j < n do
begin
  j:=(j + 1);
  if a[j] < x then
    begin
      x := a[j];
      k:= j
    end
  else
    null
end;
lemma  j = n
assert (k = least(a[i:n]) &
       x = a[k]);
a[k]:= a[i];
a[i]:= x
end

((¬ i < n &
  (TRUE &
   TRUE))
 ->
   TRUE)
# reduce
TRUE
#

((¬ j < n &
  (j <= n &
   TRUE))
 ->
  (j = n
   ->
     (k = least(a[i:j]) &
      x = a[k])))
#
lemma  j <= n
while(inner)  j < n do ...;
assert (j = n
 ->
     (k = least(a[i:j]) &
      x = a[k]))
trial invariant: (k=\text{least}(a[i:j]) \ & \ x=a[k])

\[
((k = \text{least}(a[i:j]) \ & \ x = a[k])
\rightarrow
TRUE)
\]
# reduce
TRUE
#

\[
((- \ j < n \ & 
(j <= n \ & 
(k = \text{least}(a[i:j]) \ & 
\ x = a[k])))
\rightarrow
(j = n
\rightarrow
(k = \text{least}(a[i:j]) \ & 
\ x = a[k])))
\]
# reduce
TRUE
#

\[
((j < n \ & 
(j <= n \ & 
(k = \text{least}(a[i:j]) \ & 
\ x = a[k])))
\rightarrow
(if \ a[(j + 1)] < x
\ then((j + 1) = \text{least}(a[i:(j + 1)]) \ & 
\ a[(j + 1)] = a[(j + 1)])
\ else(k = \text{least}(a[i:(j + 1)]) \ & 
\ x = a[k])))
\]
# axiom if k=\text{least}(a[i:j]) then-
\text{least}(a[i:(j+1)]) is (if a[(j+1)]<a[k] then (j+1) else k)
# reduce
TRUE
#

\[
((i < n \ & 
(TRUE \ & 
\TRUE))
\rightarrow
(((i + 1) = \text{least}(a[(i + 1):(i + 1)]) \ & 
\ a[(i + 1)] = a[(i + 1)]))
\]
# reduce
(i < n
\rightarrow
((i + 1) = \text{least}(a[(i + 1):(i + 1)])))
# set z=(i+1)
# display
\[(z = (i + 1) \rightarrow (i < n \rightarrow z = \text{least}(a[z:z])))\]

# axiom least(a[z:z]) is z
# reduce
TRUE
#
\[(l <= n \rightarrow \text{TRUE})\]
# reduce
TRUE
#
assertions verified.
output: slect2

The third pass verifies that at the end of the program, \(i = n\). Like the first pass, this pass is uninteresting. Thus, its transcript does not appear below. The fourth pass, however, succeeds in verifying that the array \(a[l:n]\) is sorted in ascending order at the end of the program. The transcript for the fourth pass follows.

input: slect2

entry \(1 <= n;\)
i := 0;
lemma \(i <= n\)
while(outer) \(i < n\) do begin
  \(i := (i + 1);\)
  \(j := i;\)
  \(x := a[i];\)
  \(k := i;\)
  lemma \((j <= n \& (k = \text{least}(a[i:j]) \& (x = a[k] \& i <= n)))\)
while(inner) \(j < n\) do begin
  \(j := (j + 1);\)
  if \(a[j] < x\) then begin

x := a[j];
k := j
end
else
null
end;
lemma (j = n &
  (k = least(a[i:n]) &
   x = a[k]));
a[k] := a[i];
a[i] := x
end;
lemma i = n
assert issorted(a[l:n])

(! i < n &
  (i <= n &
   TRUE))
->
  (i = n
   ->
   issorted(a[l:i])))
#

lemma i <= n
while(outer) i < n do ...;
assert (i = n
   ->
   issorted(a[l:i]))

trial invariant: issorted(a[l:i])
(issorted(a[l:i])
  ->
  TRUE)
# reduce
TRUE
#
#

(! i < n &
  (i <= n &
   issorted(a[l:i])))
->
  (i = n
   ->
   issorted(a[l:i])))
# reduce
TRUE
#
(((¬ j < n &
  ((j <= n &
    (k = least(a[i:j]) &
     (x = a[k] &
      i <= n)))) &
   TRUE))
  ¬>
  (((j = n &
      (k = least(a[i:j]) &
       x = a[k]))
   ¬>
    issorted(a[k <- a[i]][i <- x][l:i])))

#
lemma (j <= n &
  (k = least(a[i:j]) &
   (x = a[k] &
    i <= n))
while(inner) j < n do ...;
assert (((j = n &
      (k = least(a[i:j]) &
       x = a[k]))
  ¬>
    issorted(a[k <- a[i]][i <- x][l:i])))

trial invariant: (part(a[l:n],(i-1)) &-
issorted(a[l : (i - 1)]))

((part(a[l:n],(i - 1)) &
  issorted(a[l : (i - 1)]))
  ¬>
  TRUE)
# reduce
  TRUE
#

((¬ j < n &
  ((j <= n &
    (k = least(a[i:j]) &
     (x = a[k] &
      i <= n))) &
    (part(a[l:n],(i - 1)) &
     issorted(a[l : (i - 1)]))))
  ¬>
  (((j = n &
      (k = least(a[i:j]) &
       x = a[k]))
  ¬>
    issorted(a[k <- a[i]][i <- x][l:i])))

# axiom if x=a[k] then-
\( a[k<-a[i]][i<-x] \) is exch\((a,i,k)\)

# reduce

\[
\begin{align*}
((n <= j & \\
  (n = j & \\
  (k = \text{least}(a[i:n]) & \\
  (x = a[k] & \\
  (i <= n & \\
  (\text{part}(a[1:n],(i - 1)) & \\
  \text{issorted}(a[1 :(i - 1)])))))) \\
\rightarrow \\
\text{issorted}(\text{exch}(a, i, k)[1:i])
\end{align*}
\]

# axiom if \( \text{part}(a[1:n],(i-1)) & k=\text{least}(a[i:n]) \) then-

\[ \text{issorted}(\text{exch}(a,i,k)[1:i]) \text{ if } \text{issorted}(a[1:(i-1)]) \]

# reduce

TRUE

#

\[
\begin{align*}
((j < n & \\
  ((j <= n & \\
  (k = \text{least}(a[i:j]) & \\
  (x = a[k] & \\
  i <= n))) & \\
  (\text{part}(a[1:n],(i - 1)) & \\
  \text{issorted}(a[1 :(i - 1)])))) \\
\rightarrow \\
\begin{cases}
\text{if } a[(j + 1)] < x \\
\text{then}(\text{part}(a[1:n],(i - 1)) & \\
\text{issorted}(a[1 :(i - 1)])) \\
\text{else}(\text{part}(a[1:n],(i - 1)) & \\
\text{issorted}(a[1 :(i - 1)]))
\end{cases}
\end{align*}
\]

# reduce

TRUE

#

\[
\begin{align*}
((i < n & \\
  (i <= n & \\
  \text{issorted}(a[1:i]))) \\
\rightarrow \\
(\text{part}(a[1:n],((i + 1) - 1)) & \\
\text{issorted}(a[1:((i + 1) - 1)])))
\end{align*}
\]

# axiom \(((i+1)-1)\) is \(i\)

# reduce

\[
((i < n & \\
  \text{issorted}(a[1:i]))) \\
\rightarrow \\
\text{part}(a[1:n], i)
\]

#

lemma \( i <= n \)

while(outer) \( i < n \) do ...;
The assertion \texttt{issorted(a[1:i])} is invariant if R is, where
\[ R \rightarrow ((i < n \land \texttt{issorted(a[1:i])}) \rightarrow \texttt{part(a[1:n], i)}) \]
suggest an \( R: \texttt{part(a[1:n], i)} \)

\( (\texttt{part(a[1:n], i)}) \rightarrow ((i < n \land \texttt{issorted(a[1:i])}) \rightarrow \texttt{part(a[1:n], i)}) \)

\# reduce
\texttt{TRUE}

\#

\( (((\neg i < n \land (i \leq n \land \texttt{part(a[1:n], i)})) \rightarrow \texttt{TRUE}) \)

\# reduce
\texttt{TRUE}

\#

\( (((\neg j < n \land ((j \leq n \land (k = \texttt{least(a[i:j])} \land (x = a[k] \land i \leq n)))) \land \texttt{TRUE})) \rightarrow ((j = n \land (k = \texttt{least(a[i:j])} \land x = a[k])) \rightarrow \texttt{part(a[k <- a[i]][i <- x][1:j], i)})) \)

\#

\texttt{lemma (j \leq n \land (k = \texttt{least(a[i:j])} \land (x = a[k] \land i \leq n)))}

while(inner) \( j < n \) do 

assert \( ((j = n \land (k = \texttt{least(a[i:j])} \land x = a[k]))  

\)
\[ \text{part}(a[k \leftarrow a[i])[i \leftarrow x][l:j], i) \]

trial invariant: \( \text{part}(a[l:n],(i-1)) \)
\[
\text{part}(a[l:n],(i - 1)) \\
\rightarrow \\
\text{TRUE} \\
\#	ext{ reduce} \\
\text{TRUE} \\
\#
\]
\[
((\neg j < n \land \\
(j \leq n \land \\
(k = \text{least}(a[i:j]) \land \\
(x = a[k] \land \\
i \leq n))) \land \\
\text{part}(a[l:n],(i - 1))) \\
\rightarrow \\
((j = n \land \\
(k = \text{least}(a[i:j]) \land \\
x = a[k]) \land \\
i \leq n \land \\
\text{part}(a[l:n],(i - 1)))) \\
\rightarrow \\
\text{part}(a[k \leftarrow a[i])[i \leftarrow x][l:j], i) \\
\#	ext{ axiom if } x = a[k] \text{ then-} \\
a[k\leftarrow a[i]][i\leftarrow x] \text{ is } \text{exch}(a,i,k) \\
\#	ext{ reduce} \\
((n \leq j \land \\
(n = j \land \\
(k = \text{least}(a[i:n]) \land \\
x = a[k] \land \\
i \leq n \land \\
\text{part}(a[l:n],(i - 1)))))))) \\
\rightarrow \\
\text{part}(\text{exch}(a, i, k)[l:n], i) \\
\#	ext{ axiom if } (\text{part}(a[l:n],(i-1)) \land k = \text{least}(a[i:n])) \text{ then-} \\
\text{part}(\text{exch}(a,i,k)[l:n],i) \text{ iff TRUE} \\
\#	ext{ reduce} \\
\text{TRUE} \\
\#
\]
\[
((j < n \land \\
(j \leq n \land \\
(k = \text{least}(a[i:j]) \land \\
x = a[k] \land \\
i \leq n))) \land \\
\text{part}(a[l:n],(i - 1))) \\
\rightarrow \\
(if a[(j + 1)] < x \\
\text{ then } \text{part}(a[l:n],(i - 1)) \land \\
\text{else part}(a[l:n],(i - 1))) \\
\]
# reduce
TRUE
#

((i < n &
  (i <= n &
   part(a[l:n], i)))
->
  part(a[l:n],((i + 1) - 1)))
# axiom ((i+1)-1) is i
# reduce
TRUE
#

(1 <= n
->
  part(a[l:n], 0))
# axiom part(a[l:n],0) iff TRUE
# reduce
TRUE
#

(1 <= n
->
  issorted(a[l:0]))
# axiom issorted(a[l:0]) iff TRUE
# reduce
TRUE
#
Assertions verified.
output: slect3

The program written to file slect3 is:

entry  l <= n;
i:= 0;
lemma ((i <= n &
  part(a[l:n], i)) &
  issorted(a[l:i]))
while(outer) i < n do
begin
  i:=(i + 1);
j:= i;
x:= a[i];
k:= i;
lemma (((j <= n &
    (k = least(a[i:j]) &
     (x = a[k] &
      i <= n)) &
    part(a[l:n],(i - 1))) &
   (part(a[l:n],(i - 1)) &
issorted(a[1:(i - 1)]))
while (inner) j < n do
begin
    j := (j + 1);
    if a[j] < x then
        begin
            x := a[j];
            k := j
        end
    else
        null
end;
lemma (j = n &
    (k = least(a[i:n]) &
     x = a[k]));
a[k] := a[i];
a[i] := x
end;
lemma (i = n &
    issorted(a[1:n]))

Minimal Spanning Tree Program

The incremental verifier has also been applied to a program that computes a minimal spanning tree for connected graphs having arcs of equal cost. The algorithm underlying the program is essentially the minimum-cost spanning tree algorithm given by Aho, Hopcroft and Ullman (1976). This algorithm formulates the spanning tree problem for graphs in terms of set operations. The input graph is represented by an ordered pair (V, E), where V is a set of vertices and E a set of edges. Each edge is an ordered pair (v, w), where v and w are vertices, i.e., elements of V. The algorithm works by transforming a spanning forest for (V, E) into a single spanning tree. VS maintains the vertex sets of the trees in the spanning forest. At the start of the program,
the forest consists entirely of single-vertex trees; at the end of the program, the forest consists of exactly one tree, which spans the entire graph. T is used to collect the edges of the final spanning tree, so that the final spanning tree is \((V, T)\). The Aho et al. algorithm, slightly modified for arcs of equal cost, follows:

BEGIN
  set \(T\) to the empty set;
  set \(VS\) to the empty set;
  FOR each vertex \(v\) in \(V\) DO
    add the singleton set \({v}\) to \(VS\);
  WHILE the number of sets in \(VS\) > 1 DO
    BEGIN
      choose an edge \((v, w)\) in \(E\);
      delete \((v, w)\) from \(E\);
      IF \(v\) and \(w\) are in different sets \(W1\) and \(W2\) in \(VS\) THEN
        BEGIN
          replace \(W1\) and \(W2\) in \(VS\) by the union of \(W1\) and \(W2\);
          add \((v, w)\) to \(T\)
        END
    END
  END
END

The data structuring facilities in L are limited to single-dimensioned arrays. Thus, implementing the above algorithm in L requires that the sets and ordered pairs of the algorithm be simulated somehow by single-dimensioned arrays. In the program, the graph is assumed to have \(m\) vertices and \(n\) edges. The vertices are identified by the integers \(1\) through \(m\). The edges are specified by the arrays head[1:n] and tail[1:n], whose elements range over the vertex numbers. Thus, the \(i\)th edge is the ordered pair...
(head\[i\],tail\[i\])}. The edges of the final spanning tree are collected in head[1:newedge] and tail[1:newedge]. The array heap[1:m] maintains the vertex sets of the trees in the spanning forest: heap[i]=heap[j] if, and only if, i and j are vertices in the same tree in the forest. The minimal spanning tree program follows.

entry isconnected(m,n,head,tail);
newedges:=0;
heaps:=m;
e:=0;
while(outer) heaps>1 do
begin
  e:=(e+1);
  if heap[head[e]]\=heap[tail[e]] then
  begin
    h:=heap[head[e]];
    t:=heap[tail[e]];
    i:=0;
    while(inner) i<m do
    begin
      i:=(i+1);
      if heap[i]=t
        then heap[i]:=h
        else null
    end;
    heaps:=(heaps-1);
    newedges:=(newedges+1);
  if newedges\=e then
  begin
    head[newedges]:=head[e];
    tail[newedges]:=tail[e]
  end
  else null
  end
end

Let V be the set \{1,...,m\} and E be the set of ordered pairs \{(head[1],tail[1]),..., (head[n],tail[n])\}, where each head[i] and tail[i] \(1\leq i \leq n\) lies between 1 and m,
inclusive. Then the predicate isconnected is true if the graph \((V, E)\) is connected.
REFERENCES


180


Guttag, John V. The Specification and Application to Programming of Abstract Data Types, Technical Report CSRG-59, Department of Computer Science and Department of Electrical Engineering, University of Toronto, September 1975.

Guttag, John V., Ellis Horowitz and David R. Musser. Abstract Data Types and Software Validation, ISI/RR-76-48, Information Sciences Institute, Marina del Rey, California, August 1976.


Scott, Dana. A Type Theoretical Alternative to ISWIM, CUCH, OWHY, Unpublished notes, Oxford University, 1969.


