Formal Methods For Verification Based Software Inspection

Author

Powell, Daniel

Published

2003

Thesis Type

Thesis (PhD Doctorate)

School

School of Computing and Information Technology

DOI

10.25904/1912/1574

Formal Methods For Verification Based Software Inspection

by

Daniel Powell

BInf, Griffith University

Submitted in fulfilment of the requirements of the degree of Doctor of Philosophy to the School of Computing and Information Technology, Faculty of Engineering and Information Technology of Griffith University

December 2002

Abstract

Useful processes, that are independently repeatable, are utilised in all branches of science and traditional engineering disciplines but seldom in software engineering. This is particularly so with processes used for detection and correction of defects in software systems.

Code inspection, as introduced by Michael Fagan at IBM in the mid 1970's is widely recognised as an effective technique for finding defects in software. Despite its reputation, code inspection, as it is currently practiced, is not a strictly repeatable process. This is due to the problems faced by inspectors when they attempt to paraphrase the complicated semantics of a unit of computer code. Verification based software inspection, as advocated by the cleanroom software engineering community, requires that arguments of correctness be formulated with the code and its specification. These arguments rely on the reader being able to extract the semantics from the code.

This thesis addresses the requirement for an independently repeatable, scalable and substantially automated method for yielding semantics from computer code in a complete, unambiguous and consistent manner in order to facilitate, and make repeatable, verification based code inspection.

Current literature regarding the use of code inspection for verification of software is surveyed. Empirical studies are referenced, comparing inspection to software testing and program proof. Current uses of formal methods in software engineering will be discussed, with particular reference to formal method applications in verification.

Forming the basis of the presented method is a systematic, and hence repeatable, approach to the derivation of program semantics. The theories and techniques proposed for deriving semantics from program code extend current algorithmic and heuristic techniques for deriving invariants. Additionally, the techniques introduced yield weaker forms of invariant information which are also useful for verification, defect detection and correction. Methods for using these weaker invariant forms, and tools to support these methods, are introduced.

Algorithmic and heuristic techniques for investigating loop progress and termination are also introduced. Some of these techniques have been automated in supporting tools, and hence, the resulting defects can be repeatably identified.

Throughout this thesis a strong emphasis is placed on describing implementable algorithms to realise the derivation techniques discussed. A number of these algorithms are implemented in a tool to support the application of the verification methods presented.

The techniques and tools presented in this thesis are well suited, but not limited to, supporting rigorous methods of defect detection as well as formal and

semi-formal reasoning of correctness. The automation of these techniques in tools to support practical, formal code reading and correctness argument will assist in addressing the needs of trusted component technologies and the general requirement for quality in software.

Contents

List of Tables

List of Figures

List of Algorithms

List of Programs

Statement of originality

This work has not previously been submitted for a degree or diploma in any university. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made in the thesis itself.

Daniel Powell

Acknowledgments

First of all, I wish to express my gratitude to my supervisors, Professor Geoff Dromey and Associate Professor David Billington. Their guidance, continuous encouragement and meticulous review of my work have been invaluable in producing this thesis. I also wish to thank Geoff for the initial opportunity to return to Griffith University to undertake the work on this thesis. Special thanks also to David for his comment and advice on the more formal aspects of my thesis.

I am grateful to the School of Computing and Information Technology and the School of Information Technology of Griffith University who have provided me with scholarships and teaching opportunities which have provided financial support enabling me to work on this thesis without fear of starvation.

I wish to express my thanks to my family and friends. Without their support and encouragement, this thesis would never have eventuated.

Finally to my wife, Julie, for her love, encouragement, patience and understanding, particularly throughout the final, trying stages.

Thank you to all who shared in my effort. This would not have been possible without you.

Chapter 1

Introduction

Defective software is wide spread. This adds significantly to the cost of software and in extreme cases can lead to large financial losses and even the loss of life. A sample of recent, serious and costly software system failures includes: [42, 33, 43]

- The complete system failure of the London Ambulance Service dispatch system after only two days of operation.
- The accidental transfer of £2 billion to UK and US accounts by a UK bank's computer system.
- The Mars Climate Orbiter and Mars Polar Lander failures, where defective software resulted in mission failure at a combined cost of over $4 billion.
- The failure of the patriot missile defence system during the gulf war, resulting in the loss of 28 lives.
- The Arian 5 rocket, which self-destructed after about 35 seconds of flight due to an unhandled exception in an unnecessary module, costing half a billion dollars.

Purchasers of software are no longer willing to accept defective software. The demand for quality software, developed and verified using rigorous engineering techniques is now evident in all fields reliant on software systems. This demand is driven by the monetary cost of failing to detect and correct defects as early as possible in the software life-cycle. The relationship between the length of time a defect evades detection and the cost of locating and repairing it has long been known[12]. Generally, the longer a defect evades detection, the more expensive it is to locate and repair. Figure 1.1 shows this relationship.

The problems experienced by the software development industry regarding the cost and effort involved with achieving quality, reliable software has given rise to the term '*Software Crisis*', emphasising the flaws in the current development and verification processes and procedures.

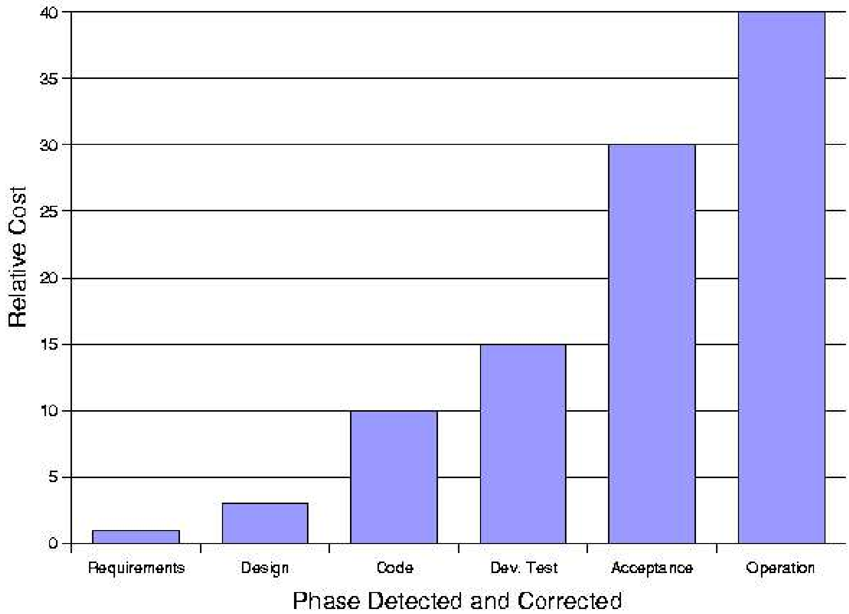


Figure 1.1: The relationship between the stage at which a defect is detected and the cost of its correction (From an analysis of 63 projects [12]).

Complementing new software developments, many software development companies are faced with the issue of verifying, validating, maintaining and reusing large amounts of legacy code which has not been subject to rigorous validation and verification. With legacy software, acceptable verification practices conforming to ISO 9001 are difficult to achieve [79].

There are a number of approaches to the verification (see Sect. 2.1) of software. These approaches can be classified as being either *proof of correctness* or *defect detection* techniques, depending on the type of verification activity. Proof of correctness techniques require the construction of arguments of correctness reasoning that the software artifact being verified satisfies some given specification. Defect detection techniques involve finding counter examples demonstrating that the software artifact deviates from its given specification.

Testing (see Sect. 2.1.2) is a defect detection technique that involves executing a given software artifact under a predefined set of conditions in an attempt to find cases where the software artifact displays a behaviour indicating a deviation from its specification. In general, software testing can not be used to demonstrate correctness, only incorrectness.

Formal verification (see Sect. 2.1.1) is a technique involving the construction of a formal argument of correctness based on mathematical principles in a formal system. Such approaches are reliant on the provision of a formal specification for the artifact being verified.

Software inspection (see Sect. 2.1.3) is a process that can be either proof or defect detection based. *Verification* based software inspection [29] involves formal verification within the inspection process framework. It is, therefore, reliant on the provision of a formal specification. The majority of practiced inspections are not formal in the sense that they involve arguments of correctness. Most software inspections are defect detection techniques that involve the application of a checklist to identify defects of certain (commonly occurring) defect types. Although most data indicates that inspections improve quality and reduce cost, checklist based inspections tend to find defects relating to style and not functionality.

In formal verification and all forms of software inspections, to find functional defects, the behavioural specification exhibited by a software artifact must be extracted from that artifact and compared to its intended specification. In this thesis we present techniques for deriving semantic assertions from a software artifact. These semantics represent the abstracted behavioural specification required to support formal verification and software inspection activities on that artifact. The repeatable techniques presented form a basis for reasoning about functional correctness and for assisting in the detection of functional defects. The derived semantics serve different purposes depending on the formality of the specification given.

Although the semantic derivation techniques are manually applied to examples throughout this thesis, we place an emphasis on the definition of algorithms for extracting semantics that are amenable to automation. We demonstrate the achievement of this goal by presenting a prototype tool and referring to sample output from this tool.

1.1 Hypothesis and Contribution

This dissertation defends the following three-part thesis:

1. Current methods of verification and defect detection are, generally, not repeatable.

2. Algorithms for the derivation of certain semantic representations from program code exist and can be mechanised.

3. The derivation of semantic information, including invariants, from program code can support and improve the repeatability of verification and inspection tasks.

The first part of this thesis is supported in the next chapter. The remaining parts are supported in subsequent chapters by defining largely mechanisable techniques for the derivation of semantic information, including invariants, from program code.

The contribution of this research includes the following:

1. The redefinition of the predicate calculus strongest postcondition predicate transformer (chapter 3) to a form which can be mechanised and which does not rely on the calculation of inverse functions or on the introduction of new variables. The lack of such a definition has previously hindered efforts to automate derivation techniques based on the strongest postcondition predicate transformer.

2. The notion of an iterative-form loop invariant (chapter 4) derivable from any loop, a mechanisable technique for the computation of this iterative invariant, and a method for its use in assisting inspection of certain classes of programs and for supporting verification by mathematical proof by induction and function-theoretic verification (chapter 9). The usefulness of this form of invariant has previously been under-estimated and techniques for its calculation, largely ignored.

3. The extension of algorithmic techniques [62] for generating invariants from loops with non-branching loop bodies, concentrating on solvable difference equations in the iterative-form invariant and systematic approaches to deriving array loop invariants (chapter 5). The techniques presented in this dissertation for non-branching loops are mechanisable.

4. The extension of algorithmic and heuristic techniques [62, 80] for the generation of invariants from loops with multiply-branching loop bodies by applying heuristic algorithms to transform the iterative-form invariant (chapter 6).

5. The definition of a mechanisable technique for deriving loop progress and termination conditions (chapter 7) directly from the code of the loop, as well as a process using the results to support verification and defect correction. As opposed to the majority of termination investigation approaches, this approach is not reliant on the provision of a formal specification or the creation of test cases, and is constructive in that the results are useful for defect correction and reuse.

6. The definition of an algorithmic approach for the abstraction of procedure semantics (chapter 8). This approach has benefits over existing techniques, in that it derives an abstracted procedure semantics which can be used for determining the semantic effects of procedure calls by simple substitution.

7. The description of verification methods based on the derived semantic forms (chapter 9).

8. The construction of an extensible prototype implementation to demonstrate the ability to mechanise many of the algorithms presented in the dissertation.

In summary, this dissertation addresses the issue of repeatability and practicality in verification and software inspection activities by proposing a formal and repeatable method of paraphrasing code into its semantics, and by defining methods for using the extracted semantics in support of these activities.

A major goal of this research was to formulate a method of deriving this semantic information which was repeatable in the sense that it relied on as little human input as possible. To this end, we have defined mechanisable, algorithmic and heuristic techniques for deriving semantics from programs. We have developed a prototype tool as part of this research which implements some of these algorithms to prove the value of the approach. We argue that the eventual automation of these techniques is necessary to support repeatable and practical methods of verification and inspection.

The generation of semantic descriptions of programs, and the detection of certain classes of defect based on these semantics, provides a number of benefits to current verification and inspection techniques

We remove the burden from the verifier in paraphrasing a code artifact into its semantic equivalent. This is an essential step in being able to reason about the correctness of the code. The method described in this dissertation produces semantic descriptions of code through a repeatable process. Because it is repeatable, the variation in inspection outcomes arising from the differences in reader understanding of code is minimised. If a formal specification accompanies the program, the derived semantics can be used as a basis for a formal proof of correctness. If an informal specification accompanies the code, the derived semantics can be useful for semi-formally arguing about correctness and for defect detection. The derivation process is not reliant on the existence of any formal documentation for the program being inspected. It is, therefore, as applicable to legacy code as to code produced, and accompanied by, a formal specification.

We illustrate the benefit of this method for supporting the detection of functional defects with the following example. Consider the Naur text reformatting problem with the following informal specification.

1. Each output line shall consist of 30 characters.
2. Each output line shall contain as many whole words as possible
3. No word shall be split across lines in the output
4. Words in the input shall be separated by one or more blanks (BL) or line feed (LF) characters. Words in the output shall be separated by only one blank and each line is ended with a linefeed character.
5. An error message shall be output if a word is more than 30 characters long.
6. A file shall be terminated by the EOT character

Without formalising this specification, we can't apply formal verification techniques, as we have no formally specified postcondition from which to proceed. However, we illustrate in chapter 4, that we can derive semantic information from Program 1, intended to meet this specification, yielding the following input-output mapping.

Ψ **(Input)**	Ω **(Output)**	**Conditions**
EOT	$BL + buf\,[1..bp_k]$	$fill_k + bp_k + 1 \leq MAXPOS$
BL	$BL + buf\,[1..bp_k]$	$fill_k + bp_k + 1 \leq MAXPOS$
LF	$BL + buf\,[1..bp_k]$	$fill_k + bp_k + 1 \leq MAXPOS$
EOT	$LF + buf\,[1..bp_k]$	$fill_k + bp_k + 1 > MAXPOS$
BL	$LF + buf\,[1..bp_k]$	$fill_k + bp_k + 1 > MAXPOS$
LF	$LF + buf\,[1..bp_k]$	$fill_k + bp_k + 1 > MAXPOS$
PC	*Line Too Long*	$bp_k = MAXPOS$
PC	*	$bp_k \neq MAXPOS$

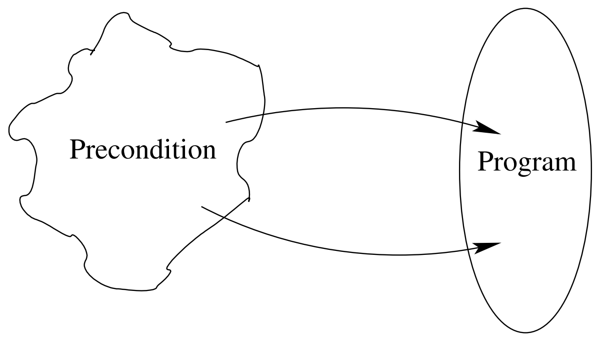
*When $bp_k \neq MAXPOS$, no output is generated and $buf\,[bp_k + 1] = c_{k+1}$ is maintained. That is, the character read is written to the word buffer, buf. PC represents a character that is not a `LF`, `BL` or `EOT` character.

Using this information we can readily reason that the program being analysed is not correct with respect to the informal specification given. A number of defects can be identified by examining the input-output mapping above. A similar table, derived automatically from the program of Fig 1 using a prototype tool, is presented in Fig. 4.3.

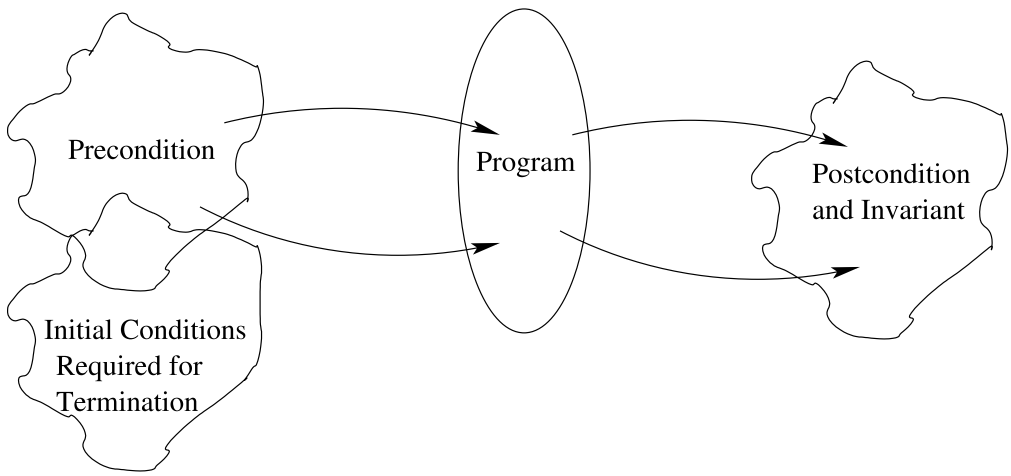
The method presented in this dissertation introduces techniques for systematically detecting certain types of defects. In certain classes of programs, loop inconsistencies in maintaining invariants, loop progress and termination failure conditions, over and under-iteration defects, and a number of quality defects are systematically found by applying the method. This means that no additional testing or inspection is required to find these defects. We have automated the generation of loop progress conditions in the prototype tool to illustrate the benefit of these techniques.

Furthermore, application of the method results in documentation that can be useful for supporting the maintenance, re-engineering and reuse of the code in question. Derived semantics provide particularly useful guidance when considering the maintenance and re-engineering of legacy code, in which the meaning of a program often needs to be rediscovered by reasoning about the program itself. Noting that a program can't be reused without a specification of what it will achieve and what it requires to achieve that, the derived semantics provides a contract [74] for reuse. Figure 1.2 summarises the specification documentation required by and produced from the semantic derivation process. The method allows us to document previously undocumented legacy code with a formal description stating its semantic purpose and describing the initial conditions required to terminate in that state.

One of the major factors impeding the practical use of formal methods in general is the perceived issue of scalability. Hall dispenses this myth in general in [50] by citing examples of industrial scale use of formal methods. However, methods such as those presented in this dissertation for extracting specifications, and particularly invariants, from programs are not directly scalable. We argue, however, that the techniques presented herein are well suited to the analysis and verification of small, well modularised units of code. The proper abstraction of requirements to a well modularised program with unit level specifications, such as is promoted in cleanroom software engineering [29, 67, 86] and the



(a) Semantic derivation requires this information.
The precondition can be true.



(b) This information is produced by semantic derivation

Figure 1.2: State space information obtained by applying the method described in this thesis.

design by contract paradigm [74], provides a means to deal with scalability. That is, scalability has to do with integration and dataflow, rather than low-level computational behaviour [19]. In chapter 8 we present a technique for the derivation of specifications from procedures which can be used to derive the semantics of procedure calls. This allows specifications for large programs to be composed from those readily derivable from their atomic procedures. In addition, as the methods presented in this dissertation rely on human reasoning, one way to deal with scalability is to define reading rules for large derived specifications or to require explicit human intervention in the derivation process. These issues of scalability are not dealt with directly by this thesis and are left for future consideration.

The techniques and methods presented in this dissertation are based on the predicate calculus 'strongest postcondition' predicate transformer and it's application in deriving strong semantic properties from code. We believe that our approach is more direct than existing methods of extracting invariants from code. In order to achieve a usable specification we have attempted to stick to a basic formal language consistent with that of the predicate calculus taught to most first year computer science students.

1.2 Thesis Organisation

The remainder of this thesis has the following organisation:

In chapter 2 we survey existing approaches and techniques for verification. Namely we discuss, the advantages and disadvantages of formal verification and formal methods, software testing, and software inspection. This is done with a view to positioning the methods of this thesis.

Chapter 3 of this dissertation presents background information on the formal methods used in the invariant extraction methods described in chapters 4, 5 and 6.

Chapter 4 describes an algorithmic technique for the derivation of an iterative-form invariant from the code of iterative program structures. It describes the use of this invariant for supporting defect detection activities.

Chapter 5 describes a technique for deriving a non-iterative-form invariant from loops with non-branching loop bodies, using the iterative-form invariant and difference equation theory.

Chapter 6 extends the techniques of chapter 5 to calculate conjunctive invariants for certain classes of loops with branched loop bodies. As a consequence of applying the algorithms in chapters 5 and 6, certain types of defects are automatically detected. These defects are discussed as they are encountered.

Chapter 7 defines rigorous techniques for determining and describing the conditions under which a loop will progress and terminate, and provides a method for using this information for verification, re-engineering and reuse.

Chapter 8 describes an algorithmic approach for abstracting the semantics of procedures and functions, and for using these extracted semantics for determining the semantic effects of procedure calls.

Chapter 9 discusses techniques for using the derived semantic information for verification by the methods of mathematical induction, direct proof, and function theoretic verification.

As discussed in the next chapter, the application of formal methods, such as the derivation techniques presented in this thesis, are difficult and complex, and, therefore, prone to errors. In light of this, this dissertation describes algorithms that lend themselves to automation. Chapter 10 discusses a prototype tool which automates some of the presented algorithms. It has been developed to prove the viability of the algorithmic and heuristic approach taken. An example of verifying a non-trivial procedure using output generated by the prototype is presented and discussed.

Conclusions and future work are discussed in chapter 11.

An overview of the methods and techniques presented in chapters 4 and 5 has been presented as a refereed paper [85] at the 9th Asia Pacific Software Engineering Conference.

Chapter 2

Background

This chapter provides the motivation for the study of an independently repeatable, automated and scalable method of yielding semantic information to support verification and defect detection activities. To present the background for this motivation, critical evaluations of current inspection, testing and theorem proving approaches to functional verification and defect detection are given.

We begin by introducing the question of correctness and approaches to verification.

2.1 Verification and Defect Detection

We use the term *specification* with respect to a program to refer to the statement of what that program is intended to do.

Validation involves applying techniques to determine if the client's concept of the system has been captured by the requirement specification, and that the implemented system meets the user's expectations. Validation usually involves acceptance and usability testing.

Verification is the process applied within the 'formal' system of the software development life-cycle that demonstrates that one phase of the life-cycle is a complete and consistent translation of the preceding phase. Verification is defined by Deutsch [20] as an activity that assures that the results of successive steps in the software development life-cycle correctly embrace the intentions of the previous steps. In the case of the implementation stage, we are concerned with verifying that it is a complete and correct representation of its design which should have been verified as a complete and correct representation of its requirements specification.

A program is said to be *correct* if it always behaves as specified.

The relationship between validation and verification can be seen in Figure 2.1.

Fagan [32], describes a *defect* as "...an instance in which a requirement is not satisfied". A program is said to be correct if it contains no defects. As we

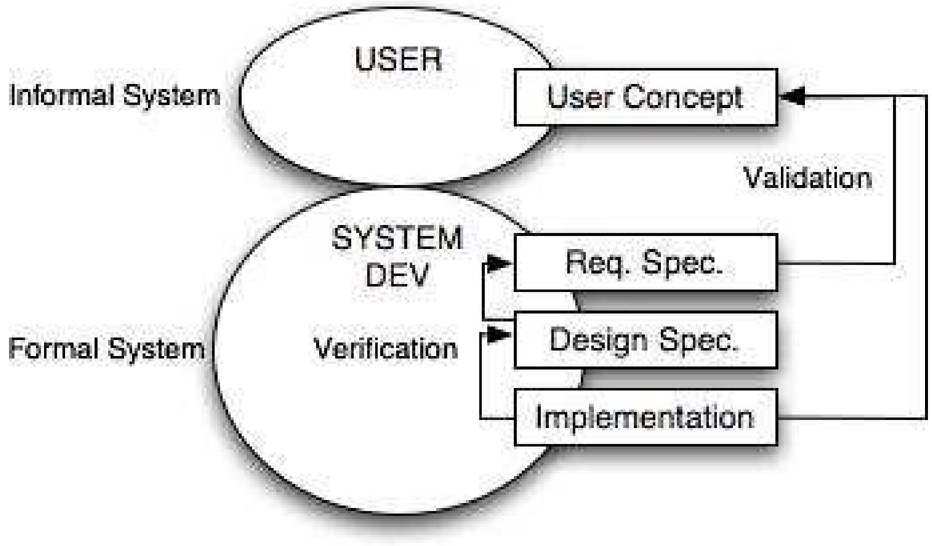


Figure 2.1: Validation and Verification and the Software Development Life Cycle

are concerned only with functional correctness in this research, if everything described by a program's specification is correctly and completely implemented by the program, then the program is correct. The task of verification is to reason about correctness.

Given a specification and a program, there are two approaches to reasoning about correctness. One approach is to find a counter-example, demonstrating that the program does not meet its specification. That is, that a program contains defects. This approach best describes the process of software *testing* and the majority of *software inspection* processes. We will refer to processes which approach verification from the point of finding counter-examples as *defect detection*.

Alternatively, we may formally reason that a program is correct with respect to its specification by way of proof. In this dissertation, we use the terms *proof of correctness*, *verification* and *formal verification* to describe a reasoning approach to demonstrating correctness. Verification is an activity that falls under the umbrella of formal methods. The proof that a unit of program code satisfies its specification comprises non-trivial activities, currently with limited mechanical support. It is not a practice that has found its way into the mainstream of software development.

This dissertation describes techniques for deriving from sequential, imperative program code, statements of the semantic effects of executing that program under any given precondition. We demonstrate how these derived semantics can then be used to support either inspection based defect detection processes, or depending on the level of specification given, formal proof of correctness. We argue that the techniques presented are repeatable and amenable to automation.

In order to position the thesis, we first critically survey current verification and defect detection processes. The next three sections describe the benefits, problems and costs associated with formal proof, testing and inspection.

2.1.1 Formal Methods and Formal Verification

'Formal Methods' is a term used in computer science to refer to a set of mathematics-based techniques, tools and notations, used for the development and verification of computer systems. Formal methods are currently applied in two ways; in the stepwise derivation of programs, and in the formal verification of computer systems.

Formal, stepwise derivation of programs [4, 23, 24, 26, 53] is concerned with applying predicate calculus transformers in a stepwise manner to a first-order specification for the program being derived. Through a sequence of stepwise logical refinements on a given precondition and postcondition, each refinement being a logical consequent of the one before it, one can derive a program that executes under the given precondition and terminates to produce the given postcondition. Software derived in such a manner, theoretically, does not need to be verified, since it was rigorously derived so that on termination it satisfies its original specification. Such methods are seldom applied in the commercial software development industry, despite their contribution to the production of functionally correct software.

Formal methods can also be applied to verification. Formal verification, as discussed here, equates to proof-based methods of verifying imperative programs. We do not consider state-exploration and model checking approaches to formal verification of finite-state, reactive systems.

Formal methods of verification are concerned with showing the functional, or semantic, correctness of a program with respect to a specification of what it should do[1].

Despite the obvious advantages of the rigour of formal methods in software derivation and verification, they are usually overlooked by industry because they are too time consuming, or simply too hard to apply. Hall [50] and Hinchey and Bowen [52] dispense some of the myths regarding formal methods.

Formal Definitions of Program Correctness

Floyd introduced the first rigorous treatment of sequential program correctness [34]. The basic premise is to annotate each statement of a program with an assertion. If the program executes under a given precondition, and if each statement of the program executes in a state described by the assertion before the

[1] This dissertation does not discuss correctness with respect to specified *safety* and *reliability* requirements.

statement resulting in an assertion that implies that given after the statement, and if the program terminates in a finite time, then the program is correct.

Hoare [53] reworked Floyd's ideas into a logical framework that gave rise to the 'Hoare triple' notation. Throughout this dissertation we use the Hoare triple as a notation to describe the semantics of program statements. Using this notation we write, $\{Q\}S\{R\}$ to mean; if condition Q holds initially, then program statement S will terminate satisfying condition R. The initial condition, Q, is referred to as the precondition of the program, S, and R is called the postcondition of S. The precondition-postcondition pair, (Q, R) is the specification for the program S. If S, executed under any state in Q, is guaranteed to establish R if it terminates, then S is said to be *partially correct* with respect to Q and R. If S can also be shown to terminate normally under all states described by Q, then S is said to be *totally correct* with respect to Q and R.

It follows that, in order to verify a program, we must show that it terminates and satisfies its specification. That is, we must show that executing a program. S, under some precondition, Q, terminates and satisfies R. This process is known as the *axiomatic approach* to verification. The axiomatic approach is a formal reasoning approach based on the following inference rules [53].

Composition: $\frac{\{Q\}S_1\{P\},\ \{P\}S_2\{R\}}{\{Q\}S_1;S_2\{R\}}$

Consequence 1: $\frac{\{Q\}S\{P\},\ P\Rightarrow R}{\{Q\}S\{R\}}$

Consequence 2: $\frac{\{P\}S\{R\},\ Q\Rightarrow P}{\{Q\}S\{R\}}$

The rule of composition simply allows us to reason about sequences of statements. The two rules of consequence allow us to reason that if S executed under some precondition Q, or a weaker precondition, establishes R or something stronger, then $\{Q\}S\{R\}$. This gives rise to two approaches for program proof of correctness; backward and forward reasoning.

Backward reasoning requires the addition of inference rules describing the programming language which allow us to determine the weakest condition, WP, that program S can be executed under and establish R. If it can then be reasoned using the second consequence rule that $Q \Rightarrow WP$, then $\{Q\}S\{R\}$, and the program must be correct. Dijkstra [24] developed the *weakest precondition* predicate transformers for the *guarded commands language* to effect the language rules of inference. Using the weakest precondition predicate transformer (wp), verification involves showing

$$Q \Rightarrow wp(S, R)$$

Forward reasoning requires the addition of inference rules describing the programming language which allow us to determine the strongest condition, SP, that program S establishes when executed under the precondition Q. If it can then be reasoned using the first consequence rule that $SP \Rightarrow R$, then $\{Q\}S\{R\}$, and the program must be correct. We use the *strongest postcondition* predicate transformers [24, 3] to effect the language rules of inference. Verification using the strongest postcondition predicate transformer (*sp*) involves showing

$$sp(Q, S) \Rightarrow R$$

The majority of formal proof approaches rely on backward reasoning. It is a goal of this research to derive information to support semi-formal human reasoning as well as formal proof. That is, we wish to derive assertions to support verification in the absence of a formal specification. The *strongest postcondition* predicate transformer, discussed in detail in the next chapter, is applicable even when a precondition is not provided (by assuming the weakest possible precondition *true*). This means that semantics can still be derived for programs with limited or no documentation, allowing comprehension and defect detection activities to be conducted. Backward reasoning techniques are reliant on strong specifications of the postcondition being provided.

Techniques for verifying programs without iterative constructs are relatively straight forward as the language inference rules and corresponding predicate transformers are easily applied and mechanised.

Iterative constructs are much more difficult to reason about. Direct methods of applying the language inference rules are, generally, impossible or infeasible to apply. Hoare's inference rule for iterative constructs involve identifying a property which remains true throughout iteration. This property is called an *invariant*.

The derivation of invariants is essential for Hoare-style proofs of correctness. This dissertation describes a number of techniques for deriving invariants and specifications to support verification activities, directly from the code.

Function-Theoretic Verification Cleanroom software development advocates function-theoretic verification. In the cleanroom correctness verification process, clear-box procedures are verified with respect to their corresponding state-box specification. This involves verifying every control structure in that clear-box procedure against its specified *intended function* [69]. In the function-theoretic approach, a procedure is correct with respect to its specification when every control structure in that procedure is verified to do what its intended function specifies. The Correctness Theorem [69] defines correctness questions for every control structure. The correctness questions for sequence, selection and iteration control structures are described in table 2.1[2]. Deck [18] and Murphy [76] assert that in practice, the formality of cleanroom verification varies depending on the formality of the specification. This dissertation defines techniques for deriving semantic information to support semi-formal and completely formal cleanroom verification. The derived semantics paraphrase the code so that the control structure correctness questions may be answered.

[2] These control structures have been translated to use the code constructs used in this dissertation.

Table 2.1: Control Structure Correctness Questions

Control Structure	**Correctness Question**
Sequence:	
[f] `g;h`	Does `g` followed by `h` do f?
Selection:	
[f] `if p` $\rightarrow$ `g [] ¬p` $\rightarrow$ `h fi`	When `p` is *true*, does `g` do f? When ¬`p` is *true*, does `h` do f?
Iteration:	
[f] `do p` $\rightarrow$ `g od`	Is termination guaranteed? Whenever `p` is true, does `g` followed by f do f? Whenever `p` is false, does doing nothing do f?

Theorem Proving This dissertation does not introduce new theorem proving techniques or tools. Rather, it introduces techniques, tools and algorithms for deriving specifications from programs to support human reasoning about correctness in verification-based and defect-detection-based software inspections. This reasoning may actually involve theorem proving on the part of the inspector. In chapter 9, we introduce a verification strategy that could be supported by existing state-of-the-art theorem provers. In this section we discuss some such theorem provers.

Theorem Provers achieve formal verification by representing systems and their properties as formulae in a mathematical logic system defined by a set of axioms and a set of inference rules. A mathematical proof is performed within this formal system to prove the properties of the system. As theorem proving is a complex task and prone to human error, tools exist that automate or assist in the proof process. Theorem provers that are most often used in the verification of software are interactive. That is, a user of the theorem prover co-operates with the prover to guide the search for a possible proof. We briefly introduce a number of interactive theorem provers here.

Among early theorem provers to support software verification were Edinburgh LCF [46] and the associated meta-language ML, and the Boyer-Moore prover based on Lisp [13] and later developed into NQTHM [14] and then ACL2[63].

Edinburgh LCF allows users to express theorems as propositions of a *theorem* type in ML [75]. Inference rules are functions from theorems to theorems. ML's secure type checking ensures that the construction of theorems is sound. It also allows theorem proving tactics, functions from goals to sub-goals, to be written in ML.

A number of general purpose theorem provers based on the foundations of Edinburgh LCF have been developed. Among those that are used for the verification of software are HOL [45] and Isabelle [83].

HOL extends LCF to support interactive theorem proving in Higher Order Logic, hence the name. There are a number of implementations of HOL. HOL88, resulting from the initial HOL implementation effort includes its own ML implementation, and is implemented on top of Common Lisp. HOL 90 and HOL 98 use only ML. HOL was initially developed as an interactive theorem prover for verification of industrial hardware, but has been successfully used to verify software.

Isabelle and Isabelle/HOL are successors to HOL. Isabelle is a generic theorem prover in which the inference rules for a logic are represented as explicit data structures not as functions[82]. It provides a framework to support proofs in a variety of logics. Isabelle has been used successfully for program proof of correctness.

Boyer and Moore's NQTHM is an interactive theorem prover based on a computational logic related to Common Lisp. The logic of NQTHM is a quantifier-free first order logic with equality. Axioms are added to the logic to describe certain data structures such as integers and natural numbers. Among NQTHM's rules of inference is a schema for proof by induction. NQTHM does not include inference rules for manipulating quantifiers, instead, relies on recursive functions to express many of the things normally expressed with quantifiers. ACL2 (A Computational Logic for Applicative Common Lisp) [15, 63] is an extended version of NQTHM intended for large scale verification projects. It uses a large applicative subset of Common Lisp rather than the original computational logic related to Lisp.

The Larch Prover (LP) [49] is an interactive theorem prover for multi-sorted first-order logic. Developed at MIT, it has been designed to work efficiently on large problems. It is used primarily for analyzing formal specifications, modelling safety-critical applications, reasoning about concurrent algorithms and hardware designs, and proving theorems in algebra. It has a number of features which are intended to make it usable for even naive users. LP carries out the routine steps of a proof automatically and attempts to provide useful information in the event that a proof fails. LP supports proof rather than automate proof.

Despite providing very high assurance that a program does as described by its specification, the majority of software systems are considered too complex for the use of theorem provers due to the amount of effort required. Theorem proving requires expertise in not only mathematics and logic, but in the language in which the theorem must be formulated and proved. The fact that the theorem prover's language does not usually resemble the language of the program being verified or the software engineer's specification language, implies that interpreting the output from a theorem prover when a proof fails is non-trivial for most software engineers. As a result of these issues, the usefulness of theorem provers and theorem proving techniques for verifying industrial software applications is still largely unexplored.

2.1.2 Testing

Software testing is the most common method of validation and verification presently used in the software industry. The idea of software testing is to exercise a program under a subset of its, possibly infinite, input domain in order to find counter examples highlighting the presence of defects. The effectiveness of testing software is related to the selection of the test cases that will be used to exercise the program. Goodenough and Gerhart [44] proposed a theory of test data selection based on the concepts of test reliability and validity, and test completeness. They introduce the notion of an ideal test, which in effect constitutes a proof of correctness for a program. They argue that properly structured tests are capable of demonstrating the absence of defects in a program. This theory forms the basis for most current test case selection methods in use today. Many authors [93, 23, 55], both in support of and against testing, have since argued that testing can only be used to detect the defects that the test was designed to find, and not to show the absence of defects in a program. We present the following argument in support of this.

Weyuker and Ostrand [93] in their refinement of Goodenough and Gerharts test data selection theory argue that the absence of defects can only be found by showing that a set of test case selection criteria is *ideal*. This involves showing that a set of test case selection criteria is *reliable* and *valid*. For a set of criteria to be reliable, every test selected by the criteria must be successful or every test selected by the criteria must fail. For a set of criteria to be valid, whenever a program is incorrect then the criteria will select at least one test set that will not successfully execute the program [44]. Weyuker and Ostrand [93] have shown that the problem with these definitions is that reliability, validity, and hence, the ideal test set are defined with respect to the entire input domain of a program. They have argued that in order to prove reliability and validity then one must already know what defects exist in the program. Under the realistic assumption that the program's defects are not known, the only way for a test selection criterion to be both reliable and valid, is if it selects the entire input domain as a test set. The entire input domain of a program is not necessarily a finite entity, and, even if it were, it would in general, be too large to be used economically for testing a program. As Howden [56] states there is no procedure, given an arbitrary program, that can produce a non-empty, finite set of test cases from the program's input domain, that whenever the program is correct on the test set it is correct on the entire input domain. As such, a program can not, in general, be shown to be correct by testing. That is, testing can not show the absence of defects in a program. For this reason, testing and test case selection has as its goal defect detection.

Weyuker and Ostrand [93] have developed a theory of test case selection that selects test cases to detect defects that fall into frequently occurring classes of defects. Much testing presently undertaken in industry meets this description. There has also been significant research into guiding the specification of tests using a program's functional specification [51, 89, 17]. This research indicates an increase in the reliability of test sets identified by analysing the properties of a program's specification. In contrast, a lot of test case selection is informal, based on experience and programmer knowledge of what defects to expect and where to expect them. Test case selection is, therefore, not independently repeatable.

Symbolic Testing: Symbolic Testing, as defined by Howden [56], is a method approximating structured testing that determines the computational effects of program constructs. It is a means to statically 'execute' a program. Symbolic evaluation focuses upon syntactic analysis of individual variable behavior, rather than modeling the semantic properties of a program. Howden demonstrates with examples, that by itself, symbolic testing finds more defects in a program than any of the other test methods defined above that select actual test data cases to test the program. Symbolic testing is effective in representing the effects of single computations but is less than effective in specifying the semantics of an entire segment of code. Symbolic testing is limited in that it makes the assumption that a program only deals with integers having an arbitrary magnitude. The execution tree resulting from a symbolic execution deals with the infinite path problem characteristic of most programs in the same way that structured testing does, by executing a path some finite number of times. Symbolic evaluation is only effective when applied to functional modules at the lowest level of abstraction [56].

It has been suggested [20] that variations in the effectiveness of software testing has a great deal to do with the experience and knowledge of the person designing the tests. As such, a lot of the testing conducted in industry is better described as an intuitive rather than a scientific process that is most successful when practiced by experienced programmers. Experienced programmers, it is claimed, know of likely places to look for defects and select test cases based on these likely defects, they are aware of 'problem modules' in the system and they understand to a much higher degree what it is the system is meant to be doing. If a programmer with knowledge and experience of the system can produce much more effective test sets than can a novice programmer, the method of test case selection used is not repeatable.

2.1.3 Software Inspection

Michael Fagan introduced the software inspection process at IBM's Kingston NY Laboratories in the early 1970s. His results were first published outside IBM in a 1976 paper [31]. The Fagan inspection method, a formalisation of review processes, is based on the application of process management techniques to software development. Inspections are formally structured peer reviews that compare deliverables in the development process with unambiguously specified correctness criteria. The aim of inspections is to prevent as many defects as possible from propagating to later stages of development, such as the test stages, where their removal is more expensive (see Fig. 1.1). Software Inspection was originally used within IBM to verify code to its design, and later modified so it could be used to verify all deliverable documents in the development life-cycle. By the early 1980s Fagan and his colleagues at IBM had come to realise that there was much benefit to be gained from modifying the method to make the most of the statistical data produced from applying inspection techniques. The changes to the original method, made by Fagan and his colleagues, were published in a paper [32]. This method has since been practiced by IBM and many other software development companies with varying but usually impressive and successful results, with most practitioners claiming an improvement in the cost and effectiveness of finding defects of one to two orders of magnitude over their previous approaches [31, 32, 42]. Fagan inspections are performed by teams with well defined roles, and generally comprise the three main steps of preparation, collection, and repair.

Most organisations using the Fagan inspection method have made slight modifications to the method, usually to the management processes employed, in order to assist in its acceptance and implementation in a variety of real world situations. The most influential change to this process was provided by Gilb and Graham [42] who introduced an extra stage to analyse the root cause of defects.

To generalise, software inspections are performed with the intent of finding and correcting defects by identifying situations where a software element deviates from the requirements specification and from the specification of it enforced by the previous step in the development life cycle, and to collect data or measures on such deviations [42, 57]. Inspections provide a means to improve the quality of the entire development project by making it possible to detect defects at any stage of the development. The economic saving on defect removal at different stages of the development (Figure 1.1) may be as great as two orders of magnitude [31, 32, 42].

The Inspection Process

The software inspection process as described in most of the literature [31, 32, 42, 57, 90] consists of at least three stages; preparation, collection, and repair. During preparation individual inspectors read the artifact separately, attempting to detect defects. In some variations of the process, these individuals are assigned certain classes of defect to focus on during the inspection phase, but are not restricted to finding and reporting this class of defect. Numerous studies [73, 92, 57] have shown that about 75% of defects are found by individuals in the preparation stage. For this reason, combined with the expense and inconvenience of organising long inspection meetings, the collection stage in many organisations simply focuses on the logging of defects. Votta [92] suggests replacing inspection meetings altogether, and tool support has been developed to assist in collecting defects without meetings. The final stage, repair, requires the author to address all of the issues raised in the inspection report. The moderator then evaluates the rework to determine if all of the reported issues have been resolved. There is generally scope for the re-inspection of work if either the moderator is not happy with the quality of the rework, or the particular artifact or deliverable is critical to the project. Gilb and Graham[42] expand this three stage process into the inspection steps; Entry, Planning, Kickoff Meeting, Individual Checking, Logging Meeting, Root Cause Analysis Edit, Follow Up, Exit.

Benefits and Costs of the Inspection Process

Gilb and Graham [42] have summarised the benefits of inspection over traditional testing methods, derived from experience data from organisations such as NASA and IBM Rochester as including:

- 30% to 100% net productivity increases;
- Overall project time saving of 10% to 30%;
- 5 to 10 times reduction in test execution costs and time;
- Reduction in maintenance costs of up to one order of magnitude;
- Improvement in consequent product quality;
- Minimal defect correction backlash at systems integration time.

These figures are comparable to others obtained by researchers such as Ackerman et al. [1] and Kitchenham et al. [64] who also claim productivity increases of up to 2 orders of magnitude in defect detection and removal, and substantial quality improvements in the amount of defects found prior to system testing.

Unfortunately, these results are not always achieved. Weighed against these benefits, is the cost of conducting inspections. Most inspections follow the Fagan or Gilb process, which requires teams of usually four or five people, trained in the inspection process. The members of these inspection teams, generally take time out from other development work to conduct inspections. Inspection meetings are often difficult to schedule and are often too short, meaning that individual inspectors often don't get enough 'air time' to raise all the issues found during the individual preparation stages.

The results of empirical studies [42] have shown that the inspection method as a whole is very effective when applied to a project from start to finish. The costs commonly associated with inspection are usually training and management costs. Software developed using inspection generally has an increased upstream cost in the verification of requirements and design, as opposed to more costly defect finding at later stages of the development.

Boehm [12] suggested that review of code is less effective than inspection of the document products of other stages. Largely, this is a result of the necessity of inspectors to paraphrase the code they are inspecting. Software code, in any non-trivial system is very likely to be extremely complex. It is often a case that inspectors will be unable to verify by inspection the very structures that, due to their complexity, are most prone to errors. Current inspection methods provide no formal rules and few tools to assist in the inspection of the complex semantics of code. The current process is not repeatable as it relies on the individual inspector's paraphrasing of the code, which, as pointed out in much of the literature [31, 32, 42, 73, 77, 84], depends largely on the experience and training of the inspectors selected, and the experience of the original author of the code. A number of authors [1, 31, 73] have stated that software inspections are best performed by inspectors who are working on, or have experience with, similar projects as they are the most likely to be able to interpret and navigate the complexities and subtleties of the inspection materials. As such, they are more likely to find more defects more efficiently than an inspector who is unfamiliar with the inspection material. Such findings support the stance that current methods of inspection are not repeatable. If such methods were repeatable, by definition, anyone applying the method would find the same defects, regardless of experience or familiarity with the project. We now cite relevant sources which examine the cause of variation in inspection outcomes. We do this to validate the stance that supporting code reading by providing semantics from which to reason about a program will improve inspection outcomes.

Variations in Inspection Outcomes

Software inspection outcomes are prone to large variation from organisation to organisation and even from project to project. There have been numerous studies [42, 57, 73, 84, 92] into determining the sources of these variations with a view to improving the outcomes of software inspection. McCarthy et al. [73] determine that the differences in the effectiveness and cost-effectiveness of software inspections come from a number of defined sources:

- the structure of the inspection process or the manner in which it is conducted;
- the inputs to the process, which include differences due to the variation in inspectors ability and experience and in the quality and complexity of the inspectables;
- the environment within which the process is conducted, including scheduling factors, workload, and organisational culture;
- the process techniques or the manner in which steps are carried out, including defect detection techniques and the use of tool support.

Most research into improving inspection has focused largely on improving the structure of the process of inspection by varying the size and number of inspection teams [10], or by changing the number or sequence of stages in inspection [42, 65]. Porter et al. [84] presents evidence that these factors and changes have little influence on the defect detection rate, but could increase the inspection interval (a measure of inspection time resource). They identified that process inputs, such as the reviewers, authors and code units had more of an effect on outcomes of inspection. Basili et al.'s outcomes [7] support these findings in determining that experienced programmers found a higher percentage of defects through code reading than through both functional and structural testing, but that the defect detection rates of students performing the same tasks did not vary between code reading, structural and functional testing. This indicates that where review methods are concerned, the experience of the reviewers has a measurable effect on defect detection rates. They also found that code reading found different types of defects (more interface defects) than did testing (more control faults).

Porter et al. [73, 84], state that the variability caused by process inputs still only accounts for a small proportion of the variation in inspection outcomes. They suggest that varying techniques of finding defects, such as changing reading techniques and using computer support may have a major effect on inspection outcomes. As this dissertation is directly related to presenting techniques for changing the defect detection techniques used in inspection, we look at results from research concerning the use of existing defect detection techniques.

Defect Detection Techniques Defect detection techniques are generally employed by individual checkers in the preparation phase of inspection. Most defect detection techniques employed in inspections are non-systematic, ad hoc or checklist procedures. Research [16, 29, 59, 79, 81] has been conducted investigating verification based inspection techniques for arguing about correctness, rather than finding defects. We refer to any inspection process that has as its primary goal the reasoning of correctness, rather than defect detection, as a *verification based inspection*.

Cleanroom Inspection Methods The cleanroom software engineering discipline [29], defines *verification based inspection* based on the work of Mills and Linger [69]. In traditional code inspection, the individuals must paraphrase the code provided to them in order to determine if it deviates from its specification. In verification based inspection, the extracted semantics are used to construct arguments to prove that a program satisfies its specification. These inspection techniques have a basis in reading by stepwise abstraction [6]. Verification-based inspection techniques have a high probability of finding functional defects if a proof fails. The sorts of defects found using such techniques are often overlooked by inspectors that employ ad hoc or checklist defect detection techniques. Cleanroom software engineering advocates that verification based inspections are conducted in place of white-box testing. Linger [68, 67] cites data from a number of projects demonstrating that verification based inspection yields a significant increase in quality and productivity over informal inspections and testing techniques. Similar results appear throughout the cleanroom literature.

Verification based inspection is, however, entirely reliant on the inspector deriving the semantics of the program being inspected in order to reason about correctness with respect to the specification. This is a non-trivial task for which cleanroom gives no guidance.

Verification Based Inspection and Module Interface Specifications In a 1994 paper [59], Jackson and Hoffman described the benefits of employing semi-formal proofs of correctness, or correctness arguments, to code being inspected based on the asserted pre and postcondition contracts described in the module interface specifications. They directed their study at industry professionals with little training in the use of formal methods, and required them to argue, in a semi-formal manner, the correctness of units of code. They found that their approach had favorable results in terms of finding functional defects, although, many working professionals had trouble presenting reasoned arguments about the code they were inspecting, even though the arguments themselves were generally not complex.

Active Design Reviews Parnas and Weiss [81] introduce a process called Active Design Reviews, in which the author poses questions to the individual reviewers to guide, or focus, their review. In order to answer the questions the reviewers are forced to develop a deep understanding of the artifact being reviewed.

Britcher [16] describes a similar process to active design reviews in which the author poses questions to the reviewers which require the construction of correctness arguments based on decomposition and refinement.

Findings from studies applying all forms of verification based inspection indicate a significant increase in finding functional, or logical, defects, as well as an overall improvement in quality, when employing verification based inspection techniques. However, they all describe the requirement for paraphrasing code to its semantics, the need for training, and a reliance on existing formal documentation for the inspection item. The lack of defined derivation processes and automated tool support means that extracting the semantics for the code being inspected is based on experience and prior understanding.

Computer Support for Inspection There are numerous products to support inspection by facilitating collaborative inspections, distributed meetings, data collection, document handling, annotation support, preparation support and metrics collection. Of these, most address process structure issues. Research [71, 70] has been done into the benefit of such tools in supporting inspection. Of the technologies that have been shown to support process techniques are preparation support, annotation support and defect detection tools. Preparation and annotation support tools provide a means by which a reader can navigate the complex structure of the code by following a sequence of hyperlinks to related information and knowledge bases, and allow the inspector to create annotations linked to the parts of the documents to which they refer.

Some products provide simple automatic defect detection such as detecting layout and coding standard violations (Code Wizard), and detecting memory leak, initialisation and array reference defects (InstantQA). These products are generally marketed as tools to find simple defects in target languages, allowing inspectors to focus on finding the more subtle defects. They are generally built around rule and knowledge bases that describe databases of implementation language specific defect types.

2.2 Other Related Work

Among other work closely related to that presented in this thesis, is that of Gannod and Cheng [37, 38, 39, 40] for using the strongest postcondition predicate transformer to derive specifications from programs to support reverse engineering. In this research the strongest postcondition predicate transformer is used to derive specifications from programs which can then be used to guide the re-engineering of a semantically equivalent program. The techniques for deriving specifications from programs without iteration and recursion in this thesis are similar to those presented by Gannod and Cheng. The techniques presented in this thesis for deriving specifications from iterative constructs could be used to extend reverse-engineering methods based on deriving specifications for imperative programs.

Extending the practice of using derived specifications for reverse engineering to the practice of *refactoring* [48, 78, 35] seems a logical step. Refactoring is defined by Fowler [35] as making changes to a body of code to improve its internal structure and maintainability, without changing its external behaviour. Refactoring is becoming an increasingly used practice of mainstream software development, and frequent refactoring is a basic tenet of Extreme Programming [8]. Refactoring within an Extreme Programming framework relies on having comprehensive test sets to be confident that a refactoring has not changed the external behaviour of a body of code. In [60], Kataoka et. al. describe a method in which invariants generated by the Daikon invariant generator [30] are used to detect candidates for particular refactorings. This work is closely related to that of [80] in which the strongest postcondition predicate transformer is used to identify candidates for quality improvements in programs. Although it is not directly addressed in this thesis, we believe that derived specifications, such as invariants, could be used to ensure the consistency of a refactoring with its original. Future research from this thesis may include the study of a method for using derived semantic information from both an original body of code and its refactoring to argue that the refactoring is semantically equivalent to the original.

2.3 Conclusion

In this chapter we have presented general background material to support and position the thesis. We have surveyed existing literature on formal methods, software inspection, testing, theorem proving, and reverse engineering and refactoring as it relates to the thesis. The lack of repeatability in current, semiformal verification activities has been raised. The lack of repeatability in code inspection, is attributed to difficulty in paraphrasing the code being inspected to extract a semantics from that code. This thesis addresses this concern and describes techniques for using the extracted semantics to support software inspection.

Chapter 3

Preliminaries

In this chapter we introduce definitions and notations used in this thesis. In particular we provide definitions for the *Strongest Postcondition*, *sp*, transformer, discuss some shortcomings of the existing definitions in terms of their ability to be automated, and suggest semantically equivalent, automatable, definitions. We assume readers are familiar with first-order logic. If they are not, we direct them to [26] for a good discussion on logic for program design.

3.1 Terms, Formulas and their Functions

For the expressions of ℓ, a first-order language, we define terms, formulas and quantifier free formulas.

Definition: Term

A *term* can be constructed by a finite number of applications of the following formation rules:

1. All variable symbols of ℓ are terms.
2. All constant symbols of ℓ are terms.
3. If $t_1, t_2, ..., t_n$ are terms of ℓ and f is an n-ary function symbol then $f(t_1, t_2, ..., t_n)$ where $n \geq 1$ is also a term.

Definition: Atomic Formula

An expression of ℓ is an *Atomic Formula* iff it is of the form $F(t_1, t_2, ..., t_n)$, where F is an n-ary predicate symbol, and $t_1, t_2, ..., t_n$ are terms of ℓ, and $n \geq 1$.

Notation: Logical Connectives

We use the following connectives and symbols, where p and q denote propositions:

Connective	Usage	Symbolic Form
not	not p	$\neg p$
and	p and q	$p \wedge q$
or	p or q	$p \vee q$
if p then q	p implies q	$p \Rightarrow q$
p if and only if q	p equivalent to q	$p \Leftrightarrow q$

The following left to right order of precedence is used throughout this dissertation to reduce the required number of parentheses for compound propositions:

$$\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$$

For example, $(p \wedge q) \vee (\neg r)$ becomes $p \wedge q \vee \neg r$. Parentheses are used in this dissertation where it makes reading simpler.

Definition: Formula

All *formulas* can be constructed by a finite number of applications of the following formation rules:

1. An atomic formula is a formula.
2. If F is a formula, then $(\neg F)$ is a formula.
3. If F and G are formulas, then $(F \wedge G)$, $(F \vee G)$, $(F \Rightarrow G)$, and $(F \Leftrightarrow G)$ are also formulas.
4. If F is a formula then $(\forall x\,(F))$ and $(\exists x\,(F))$ are also formulas.

Notation: Confined Quantification

We will use $(\forall x \in S\,(F))$ to mean $(\forall x(x \in S \Rightarrow F))$ and $(\exists x \in S\,(F))$ to mean $(\exists x(x \in S \wedge F))$.

Definition: Quantifier Free Formula (qff)

All *quantifier free formulas* can be constructed by a finite number of applications of the following rules:

1. An atomic formula is a qff.
2. If F is a qff, then $(\neg F)$ is a qff.
3. If F and G are qffs, then $(F \wedge G)$, $(F \vee G)$, $(F \Rightarrow G)$, and $(F \Leftrightarrow G)$ are also qffs.

Definition: The free-variable function

The free-variable function, $\mathbf{V}$, takes as its argument, F, a formula, term or program, and returns the set of free variables contained in F, such that:

$\mathbf{V}(F) = \{x | x \text{ occurs in } F \text{ and } x \text{ is a free variable}\}$

For example, $\mathbf{V}(x > 3 \wedge y \leq x) = \{x, y\}$.

Definition: The substitution functions[1]

Given any term or formula, F:

- $F[y/x]$ denotes the formula obtained from F by replacing all *free* occurrences of x in F by y.
- $F_{all}[y/x]$ denotes the formula obtained from F by replacing *all* occurrences of x in F by y.
- $F_{bd}[y/x]$ denotes the formula obtained from F by replacing all *bound* occurrences of x in F by y.

For Example: $(x = z \wedge y \leq x)\,[z/x] \equiv (z = z \wedge y \leq z) \equiv (y \leq z)$.

Notation: Set Comprehension

The following set comprehension is used throughout this thesis:

- $\{x_1, \ldots, x_n\}$ is an extensional specification of a set containing exactly the elements listed by $x_1, \ldots, x_n$. For example, $\{1, 2, 3, 4\}$ represents the set containing exactly $1, 2, 3$ and 4.
- $\{e \in T \mid P\}$ is a comprehensive specification for the set containing those e in T satisfying P. For example, $\{e \in \mathbb{Z} \mid 0 < e < 5\}$ specifies the set of all integers between and including 1 and 4.
- $\{e \in T \bullet exp(e)\}$ is a comprehensive specification for the set of values obtained by applying the expression exp to all e in T. For example, $\{e \in \mathbb{N} \bullet e^2\}$ specifies the set of the squares of all natural numbers. The elements in the final set may not necessarily be elements of T.
- $\{e \in T \mid P \bullet exp(e)\}$ is a comprehensive specification for the set of values obtained by applying the expression exp to all e in T which satisfy P. For example, $\{e \in \mathbb{N} \mid e < 5 \bullet e^2\}$ specifies the set of the squares of all natural numbers less than 5. That is, $\{0, 1, 4, 9, 16\}$.

[1] We have decided to use the notation, $F[y/x]$, for the substitution functions rather than using other common notations like $F[x := y]$ or $(x := y).F$, because the use of ":=" for substitution is confusing in our context of verifying code units where ":=" is often used as an assignment operator.

Notation: Special Sets

We use the following notations to represent commonly used sets:

- $\{\}$ represents the empty set.
- $\mathbb{Z}$ represents the set of integers.
- $\mathbb{N}$ represents the set of natural numbers. That is, the set of integers greater than or equal to zero, $\{0, 1, 2, \ldots\}$.
- $\mathbb{N}^+$ represents the set of natural numbers excluding zero. That is, $\{1, 2, \ldots\}$.
- $\mathbb{R}$ represents the set of real numbers.
- $\mathbb{B}$ represents the set $\{true, false\}$.

Notation: Range Specification

$j \in [a..b]_{\mathbb{Z}}$ means $a \leq j \leq b \wedge j \in \mathbb{Z}$.
$j \in (a..b]_{\mathbb{Z}}$ means $a < j \leq b \wedge j \in \mathbb{Z}$.
$j \in [a..b)_{\mathbb{Z}}$ means $a \leq j < b \wedge j \in \mathbb{Z}$.
$j \in (a..b)_{\mathbb{Z}}$ means $a < j < b \wedge j \in \mathbb{Z}$.
$j \in [a..b]_{\mathbb{R}}$ means $a \leq j \leq b \wedge j \in \mathbb{R}$.
$j \in (a..b]_{\mathbb{R}}$ means $a < j \leq b \wedge j \in \mathbb{R}$.
$j \in [a..b)_{\mathbb{R}}$ means $a \leq j < b \wedge j \in \mathbb{R}$.
$j \in (a..b)_{\mathbb{R}}$ means $a < j < b \wedge j \in \mathbb{R}$.
The abbreviated notation $j \in [a..b]$ is used throughout to mean $j \in [a..b]_{\mathbb{Z}}$.

Notation: Sequences

We use the notation **seq** T to denote the set of all sequences over T. The empty sequence is denoted $<>$. Throughout this thesis we do not differentiate between dynamically sized and statically sized sequences. All sequences are treated as dynamic arrays. A sequence, or array, over type T is declared with `s : seq` T, which yields the semantic predicate $s \in \textbf{seq}\ T$.

Programming Langugage

For the purpose of discussing the rules for the calculation of semantic information from imperative programs, we present all examples in a language which makes available all core constructs of imperative languages. We present descriptions of code structure and code examples with a syntax resembling Guarded Commands language [21]. The syntax of this language is defined in Appendix A. We assume that arrays are indexed beginning from position 0. Deviations from Dijkstra's Guarded Command Language are defined as they are encountered.

3.2 The Strongest Postcondition Predicate Transformer

Throughout this thesis, we refer to the Hoare model of functional correctness. The Hoare triple [53] specifies the semantics of program statements. Using this notation we write, $\{Q\}S\{R\}$ to mean; if condition Q holds initially then program statement S is guaranteed to terminate satisfying condition R. The initial condition, Q, is referred to as the precondition of the program S, and R is called the postcondition of S. This triple represents a model of *total correctness*. The notation $Q\{S\}R$ represents a partial correctness model of execution. That is, $Q\{S\}R$ indicates that if the condition Q holds initially and the program S executes and terminates, then R will hold. We can visualise this model as Q and R being a set of states, and S, being a collection of mappings from elements of Q to R.

In order to verify a program we must show that it satisfies its specification and terminates. That is, we must show that a program. S, executed under some precondition, Q, terminates and satisfies a postcondition R. A program may be verified by either forward or backward reasoning by applying predicate transformers to the specifications and program [80].

Weakest Preconditions: The weakest precondition predicate transformer, $wp(S, R)$, provides the weakest definition of the set of all initial states under which the program S can execute and terminate and guarantee that some postcondition R will hold. It is used in the "backwards reasoning" approach to program verification by showing that where we have $\{Q\}S\{R\}$, $Q \Rightarrow wp(S, R)$. Dijkstra [24] provides rules for calculating the weakest precondition of a program constructed from assignment, selection and iteration constructs. Traditionally, weakest precondition calculations have provided a strong basis for the derivation of programs from a specification using a process of stepwise refinement, and have also been used as a basis for formal proof.

Strongest Postconditions: The strongest postcondition transformer, $sp(Q, S)$, gives the strongest assertion R that holds given that S executes and terminates under a precondition satisfying Q. Strongest postconditions, therefore, specify the strongest assertions that hold at each point in the execution of the program S. A program satisfies it's specification when $sp(Q, S) \Rightarrow R$. Dijkstra [24] provides rules for computing the strongest postcondition for a program consisting of assignment, selection and iteration[2] constructs. This thesis redefines and uses these rules as a basis for paraphrasing code in terms of a complete, unambiguous first-order specification of its semantics.

In this section, we introduce the notation and rules for computing the strongest postcondition of a program.

[2]The rules for iteration are only applicable if it is known in advance the number of times that the loop will iterate. This is the cause of the difficulty of generating a loop invariant. This thesis attempts to solve the loop invariant problem for at least a subset of loops.

3.2.1 Strongest Postcondition for Assignment

Dijkstra [24] defines the strongest postcondition for assignment as:

1. $sp(Q,\ x := E) \equiv (x = E) \wedge \exists x\,(Q)$, and
2. $sp(Q,\ x := E(x)) \equiv Q\left[E^{-1}(x)/x\right]$

The first definition, used when x is assigned an expression which is not a function of x, makes direct calculation difficult due to the term $\exists x\,(Q)$ only indicating that x is bound by Q [80]. The second definition may make calculation impossible in many situations as the inverse function may not be defined or may be difficult to capture $\left(x := 3x^6 - 4x^2 + 2\right)$.

Pan [80] suggested the introduction of fresh variables to the problem in order to remove the requirement to calculate inverse functions, by effectively removing any assignments of the second form. This approach highlights quality defects in code by identifying situations where a single variable is used for more than one purpose. By Pan's approach, introducing the new variable t, the calculation $sp\left(x = X,\ x := 3x^6 - 4x^2 + 2\right)$ becomes

$$
\begin{aligned}
& sp\left((x = X)\,[t/x],\ x := \left(3x^6 - 4x^2 + 2\right)[t/x]\right) \\
\equiv\ & sp\left(t = X,\ x := \left(3t^6 - 4t^2 + 2\right)\right) \equiv \left(t = X \wedge x = 3t^6 - 4t^2 + 2\right)
\end{aligned}
$$

This allows us to describes the semantics of the assignment statement at the cost of introducing variables that are not part of the original program variable set, and which can not be removed without applying the inverse function that we are trying to avoid. Because it is the aim of this thesis to present a semantic description of a unit of code to be used to assist in the inspection of the code, we feel it may be confusing to introduce variables where-ever a self-assignment is made. To this end, we provide an alternate definition of the strongest postcondition for assignment which is equivalent to the two calculations defined by Dijkstra. This definition removes the necessity to calculate inverse functions, is repeatable, and does not rely on the introduction of new variables to the problem.

The Modified Strongest Postcondition Calculation

We start with the conjecture that in any program written in an imperative language that allows self-assignment, a variable must have been initialised with a value before being used on the right hand side of an assignment. If a variable has not been initialised, prior to it being used on the right hand side of an assignment, then we can say that the assignment in question is non-deterministic. For example, if we consider the assignment $x := x^2$ under the precondition $true$, we can make no reasonable guess at what x might be in the postcondition. The value of x is determined either by the value that happens to be stored at the particular memory location allocated for x, or the language default for variables of the type of x. We denote this non-determinable state of x by x_0. The definition for the strongest postcondition for assignment provided here uses the conjecture that all variables must be initialised with a value, either explicitly by the program or implicitly by the environment, prior to their use in an expression.

Algorithm 1 *sp* for assignment

Consider the assignment $x := E$, under some precondition Q where x is a variable, E is a term, and Q is a formula of the form $Q_1 \vee ... \vee Q_n$ where the sub formulas $Q_1, \ldots, Q_n$ contain no disjuncts.
$sp(Q, x := E) = sp(Q_1, x := E) \vee .. \vee sp(Q_n, x := E)$, where for all $i \in [1..n]$, $sp\,(Q_i, x := E)$ is defined as:

1. If an xC-equality occurs in Q_i, let the constant symbol occurring in the leftmost xC-equality of Q_i be D, then $sp(Q_i, x := E) = Q_i[D/x] \wedge x = E[D/x]$.
2. If an xC-equality does not occur in Q_i, and an xv-equality occurs in Q_i, let the non-x variable occurring in the leftmost xv-equality of Q_i be w, then $sp(Q_i, x := E) = Q_i[w/x] \wedge x = E[w/x]$.
3. If an xC-equality does not occur in Q_i, and an xv-equality does not occur in Q_i, and an xf-equality occurs in Q_i, let the non-x term occurring in the leftmost xf-equality of Q_i be u, then $sp(Q_i, x := E) = Q_i[u/x] \wedge x = E[u/x]$.
4. Otherwise, x is not defined in the scope of the assignment prior to use, and $sp(Q_i, x := E) = Q_i\,[x_0/x] \wedge x = E[x_0/x]$.

Within a program we may establish a number of equalities between a variable and other variables, constants or functions of constants or variables. In order to make the strongest postcondition calculation deterministic, we distinguish between the following types of variable equalities:

Definition 1. xC-equality

An xC equality is any atom of the form $x = C$ or $C = x$ where x is a variable and C is a constant symbol.

Definition 2. xv-equality

An xv equality is any atom of the form $x = v$ or $v = x$ where x and v are variables.

Definition 3. xf-equality

An xf equality is any atom of the form $x = f(t_1, t_2, ..., t_n)$ or $f(t_1, t_2, ..., t_n) = x$ where x is a variable and f is an n-ary function symbol and $t_1, t_2, ..., t_n$ are terms.

The new *sp* calculation for assignment is described by Algorithm (1).

Example 3.1. Calculating *sp* for assignment

$$
\begin{array}{rl}
\text{(i)} & sp(x = a \wedge z = y * x, x := y) \\
\equiv & (x = a \wedge z = y * x)[a/x] \wedge x = y[a/x] \\
\equiv & (a = a \wedge z = y * a) \wedge x = y \\
\equiv & z = y * a \wedge x = y
\end{array}
$$

$$
\begin{array}{rl}
\text{(ii)} & sp\left(x = X, x := 3x^6 - 4x^2 + 2\right) \\
\equiv & (x = X)[X/x] \wedge x = \left(3x^6 - 4x^2 + 2\right)[X/x] \\
\equiv & (X = X) \wedge x = \left(3X^6 - 4X^2 + 2\right) \\
\equiv & x = \left(3X^6 - 4X^2 + 2\right)
\end{array}
$$

As a side effect of the calculation process we may achieve automatic recognition of defects that relate to the use of an unitialised variable. If the result of the *sp* calculation involves any references to v_0 for a variable v then v has been used prior to being initialised in the scope of the program segment being analysed. There are a number of reasons why this may occur:

- If v is declared locally, within the scope of the program segment being analysed, then v_0 indicates that v has not been initialised prior to use. This is a defect that effects portability, maintainability and reusability.
- If v is not declared locally then v_0 indicates that v may be either a global variable, or a class variable (in OO programs).

The identification and reporting of such semantics is useful to assist code readers in identifying defects related to variable initialisation.

It should be noted that due to the treatment given to sub-programs in chapter 8, the precondition prior to the execution of the body of a procedure or function includes an xv-equality of the form $v = v?$ for all formal parameters v. This ensures that formal parameters are not considered uninitialised.

Lemma 3.1. For a simple assignment of the form $x := ax + b$ under some precondition Q, where x is a variable, and a and b are constants ($a \neq 0$), $sp(Q, x := ax + b) = Q\left[\left(\frac{x-b}{a}\right)/x\right]$.

Proof. For the assignment $x := ax + b$ to occur correctly in an imperative program, Q must be of the form $(x = t) \wedge P$, where t is a term, otherwise x is uninitialised prior to its use in the assignment, and the assignment itself is defective.

$$
\begin{aligned}
LHS &= sp(Q, x := ax + b) \\
&= sp((x = t) \wedge P, x := ax + b) \\
&= ((x = t) \wedge P)[t/x] \wedge x = (ax + b)[t/x] \\
&= ((t = t) \wedge P[t/x]) \wedge x = (at + b) \\
&\equiv P[t/x]) \wedge x = (at + b) \\
RHS &= Q\left[\left(\frac{x-b}{a}\right)/x\right] \\
&= ((x = t) \wedge P)\left[\left(\frac{x-b}{a}\right)/x\right] \\
&= \left(\left(\left(\frac{x-b}{a}\right) = t\right) \wedge P\left[\left(\frac{x-b}{a}\right)/x\right]\right) \\
&\equiv \left(\left(\left(\frac{x-b}{a}\right) = t\right) \wedge P\,[t/x]\right) \\
&\equiv (x = at + b) \wedge P[t/x] \\
&\equiv P[t/x] \wedge (x = at + b) \\
&\equiv LHS
\end{aligned}
$$

□

sp for Sequence Element Assignment

For an assignment of the form $a[exp] := E$, executing under the precondition Q, where exp and E are expressions and a is a sequence data type, the above definition of strongest postcondition for assignment and the substitution function can be extended to describe the changes to a as well as the fact that the remainder of a is unchanged by the assignment.

We use the notation $a = A$, where a and A are sequence variables, to denote the relationship $\forall i \in [0..\#a-1]\,(a[i] = A[i])$. That is, writing $a = A$ is equivalent to $(a[0] = A[0] \wedge a[1] = A[1] \wedge \cdots \wedge a[\#a-1] = A[\#a-1])$.

The semantics of the substitution $F\,[exp/a[e]]$, where F is a term or formula, a is a sequence variable and e and exp are expressions, can be obtained by application of the following rules:

1. Evaluate e and denote the result as e'.
2. Make the substitution $F\,[exp[e'/e]\;/\;a[e']]$.

This substitution function, when applied to Algorithm 1, yields the following definition for the strongest postcondition for assignment to a sequence element.

Let $Q \equiv (P \wedge a = A)$ (A may be a_0 if a is uninitialised) and let exp' denote the evaluation of the expression exp.

Then

$$\begin{aligned}
& sp(Q,\ \texttt{a[exp]:=E}) \\
\equiv\ & P[A[exp]/a[exp]] \wedge (a = A)[A[exp]/a[exp]] \wedge a[exp] = E[A[exp]/a[exp]] \\
\equiv\ & \left(\begin{array}{l} P[A[exp']/a[exp']] \wedge \\ (a[0] = A[0] \wedge a[1] = A[1] \cdots \wedge a[\#a-1] = A[\#a-1])\,[A[exp']/a[exp']] \wedge \\ a[exp] = E[A[exp']/a[exp']] \end{array} \right) \\
\equiv\ & \left(\begin{array}{l} P[A[exp']/a[exp']] \wedge \\ A[exp'] = A[exp'] \wedge \forall i \in [0..\#a-1]\,(i \neq exp \Rightarrow a[i] = A[i]) \\ a[exp] = E[A[exp']/a[exp']] \end{array} \right) \\
\equiv\ & \left(\begin{array}{l} P[A[exp']/a[exp']] \wedge \forall i \in [0..\#a-1]\,(i \neq exp \Rightarrow a[i] = A[i]) \wedge \\ a[exp] = E[A[exp']/a[exp']] \end{array} \right) \qquad (3.1)
\end{aligned}$$

For example, the strongest postcondition semantics of the assignment `a[i]:=x` under the precondiition $(a = a? \wedge i = i? \wedge v = v?)$, where a and $a?$ are sequences and i is a natural number, is calculated by (3.1) as follows:

$$\begin{aligned}
& sp((a = a? \wedge i = i? \wedge v = v?)\,,\ \texttt{a[i]:=v}) \\
\equiv\ & (i = i? \wedge v = v? \wedge \forall j \in [0..\#a-1]\,(j \neq i \Rightarrow a[j] = a?[j]) \wedge a[i] = v[a?[i?]/a[i?]]) \\
\equiv\ & (i = i? \wedge v = v? \wedge \forall j \in [0..\#a-1]\,(j \neq i \Rightarrow a[j] = a?[j]) \wedge a[i] = v)
\end{aligned}$$

Higher Level, Non-Sequential Data Types

We define a higher-level data type as one composed of elements or attributes. For example, in a language like C, these include structures, unions and arrays. Assignment to arrays is dealt with above. We can take a similar approach to dealing with assignment to other higher level data types. For example, consider the C assignment `astNode.lineNo = yylineno;` to an element `lineNo` of some higher level data type called `astNode`. We treat this as an assignment to a component variable with the *fully qualified name*, `astNode.lineNo`. Thus, by the redefined rule for sp for assignment, the equality $astNode.lineNo = yylineno$ is yielded.

3.2.2 Strongest Postcondition for Selection

Dijkstra [24] defines the strongest postcondition for a selection statement, $if\ C_1 \rightarrow S_1\ []..[]\ C_n \rightarrow S_n\ fi$, under the precondition Q, where each C_i is a boolean expression and each S_i is a statement as:

$$sp(Q, if\ C_1 \rightarrow S_1\ []..[]\ C_n \rightarrow S_n\ fi) \ = \ sp(Q \wedge C_1, S_1) \vee ... \vee sp(Q \wedge C_n, S_n)$$

Example 3.2. Calculating sp for selection

$$\begin{aligned}
& sp(N \geq 0 \wedge a = x, if\ a \geq N \rightarrow a := a+1\ []\ a < N \rightarrow a := a+a\ fi) \\
=\ & sp(N \geq 0 \wedge a = x \wedge a \geq N, a := a+1) \vee sp(N \geq 0 \wedge a = x \wedge a < N, a := a+a) \\
=\ & (N \geq 0 \wedge a = x+1 \wedge x \geq N) \vee (N \geq 0 \wedge a = 2x \wedge x < N)
\end{aligned}$$

3.2.3 Strongest Postcondition for Sequence

Dijkstra [24] defines the strongest postcondition for sequential statements, $S_1; S_2$, under the precondition Q as:

$$sp(Q, S_1; S_2) \; = \; sp\left(sp\left(Q, S_1\right), S_2\right)$$

Example 3.3. Calculating *sp* for sequence

$$\begin{array}{ll} & sp(True, b := 1; c := 2 * b;) \\ = & sp\left(sp\left(True, b := 1\right), c := 2 * b\right) \\ = & sp\left(True \wedge b = 1, c := 2 * b\right) \\ = & (True \wedge b = 1 \wedge c = 2b) \\ = & (b = 1 \wedge c = 2b) \end{array}$$

3.2.4 Strongest Postcondition for Iteration

Back [3] and Dijkstra [24] define the strongest postcondition for the iteration `do` $G \rightarrow S$ `od`, under the precondition Q, where G is a boolean expression and S is a statement as:

$$sp(Q, do\, G \rightarrow S\; od) \; = \; (\neg G \wedge (Q \vee P_1 \vee ... \vee P_m))$$

where m is a positive integer representing the number of iterations, and $P_1 = sp(Q \wedge G, S)$ and $P_{i+1} = sp(P_i \wedge G, S)$ for $0 < i \leq m$.

The application of this definition is difficult, if not impossible, in practice, because, although the calculation depth, m, is bound, it is not fixed. This stops us from deriving the strongest postcondition for a loop unless we know in advance how many times the loop will iterate. The net effect of this is that we can not use this definition of strongest postcondition calculation to derive a specification of what a loop is doing. This impedes our verification efforts. Pan [80] introduces a process to obtain the *Strongest Accessible Conjunctive Postcondition*, in which the calculation depth depends only on the loop body and the loop precondition, and is therefore fixed. This method is limited in its application to only a small subset of simple loops.

The inability to determine the calculation depth of a loop is largely responsible for the inability to derive loop invariants and, therefore, loop postconditions through calculation alone. This fact is the motivation behind chapters 4 and 5 of this thesis.

3.2.5 Strongest Postcondition for Procedure Call

In defining the strongest postcondition for procedure calls, we must describe all direct effects and side effects of executing a procedure under a given precondition.

Hoare [54] extends the axiomatic basis for computer programming to describe an axiomatic approach for reasoning about procedures and explicitly about recursive procedures.

Gries [47] presents a theorem for specifying the semantic effects of executing a procedure call to a non-recursive procedure. This theorem provides a means by which the specification of a procedure call can be constructed by in-lining the body of the called procedure and multiple assignments representing parameter instantiation and returns in the context of the procedure call [47, 38]. That is, assuming the procedure `p` is guaranteed to terminate, a procedure call to `p` with actual *value* parameter list a, actual *reference* parameter list b and actual *result* parameter list c, and the corresponding formal parameter lists x, y and z, under the initial conditions PR

```
...
{PR}
p(a,b,c)
{R}
...
```

is syntactically equivalent to

```
...
{PR}
x,y := a,b;
{P}
< body >
{Q}
y,z := u,v;
{QR}
b,c := y,z;
{R}
...
```

where *body* is the body of the procedure `p` and u and v represent local variable lists used to compute the reference and result parameters b and c, respectively.

This definition for calculating the strongest postcondition for a procedure call is hindered by the fact that the strongest postcondition for the procedure body must be calculated every time it is called. Chapter 8 of this thesis presents an algorithm which yields an abstract specification for a procedure and an algorithm to define the effects of a procedure call to a procedure by substituting actual for formal parameters in that procedure's derived abstract specification. Therefore, the specification for the procedure is only calculated once. The substitution algorithms presented in Chapter 8 also provide a means by which a procedures precondition can be checked prior to the procedure call.

3.2.6 Strongest Postcondition for I/O Statements

Pan [80] provides a discussion on and definition for the strongest postcondition predicate transformer for I/O statements. We consider two special functions in our language, `read` and `write`, to read a variable and write an expression, to and from the standard input and output streams, respectively. We refer to [80] for a detailed discussion on the semantics of I/O statements and provide the following definitions.

Let Ψ represent the sequence of standard input, and let Ω represent the sequence of standard output, then

$$sp(Q,\ \texttt{read(x)}) \equiv sp(Q, \texttt{x:=x?;}\Psi := \Psi \frown x?) \tag{3.2}$$

and

$$sp(Q,\ \texttt{write(exp)}) \equiv sp\,(Q,\ \ \Omega := \Omega \frown exp) \tag{3.3}$$

where x is a variable, $x?$ is a newly introduced variable, exp is an expression of accessible program variables and $\frown$ is the sequence concatenation operator. These definitions allows us to construct a specification of the known values in the input and output sequences.

Example 3.4. Calculating sp for I/O

$$\begin{aligned}
& sp(x = X \wedge a = x,\ \texttt{read}(\texttt{x}); \texttt{write}(\texttt{x})) \\
= \quad & sp\,(sp\,(x = X \wedge a = x,\ \texttt{read}(\texttt{x}))\,,\ \texttt{write}(\texttt{x})) \\
= \quad & sp\,(a = X \wedge x = x? \wedge \Psi = \Psi_0 \frown x?,\ \texttt{write}(\texttt{x})) \\
= \quad & (a = X \wedge x = x? \wedge \Psi = \Psi_0 \frown x? \wedge \Omega = \Omega_0 \frown x?)
\end{aligned}$$

3.2.7 Strongest Postcondition for Variable Declaration

To be consistent and complete with respect to typed languages we define the strongest postcondition predicate transformer for variable declarations.

Throughout this thesis we represent the variable declaration of a variable v of type T by `v:T;`. We assume that in a typed language no predicates involving a variable v can exist prior to its declaration. The semantic effect of the declaration is to assert the predicate $v \in T$. That is, after declaration, v must be instantiated with some value in the set of values described by the type T. We propose the following strongest postcondition predicate transformer:

$$sp\,(Q, \texttt{v:T;}) = Q \wedge v \in T \tag{3.4}$$

If at some later stage v is assigned a value not in the set of values of type T, then the predicate $v \in T$ will be contradicted. This may be a useful method for repeatably identifying type mismatch defects in assignments and parameter substitutions[3]. This problem is addressed later in this thesis.

[3] This will only be a useful check if the intersection between distinct types is the empty set.

3.2.8 Applying the *sp* Transformers

The application of these transformers is demonstrated in the following (non-iterative) code example:

$\{n, m, t \in \mathbb{N} \wedge n = n? \wedge m = m?\}$

```
if n≤m→
    skip
[] n>m→
    t:=n; n:=m; m:=t
fi
```

Assuming S represents the program above, the strongest postcondition is calculated as: /small

$$
\begin{array}{ll}
 & sp(n, m, t \in \mathbb{N} \wedge n = n? \wedge m = m?,\ S) \\
\equiv & \left(\begin{array}{l} sp\,(n, m, t \in \mathbb{N} \wedge n = n? \wedge m = m? \wedge n \leq m,\ \mathtt{skip}) \vee \\ sp\,(n, m, t \in \mathbb{N} \wedge n = n? \wedge m = m? \wedge n > m,\ \mathtt{t} := \mathtt{n}; \mathtt{n} := \mathtt{m}; \mathtt{m} := \mathtt{t}) \end{array} \right) \\
\equiv & \left(\begin{array}{l} (n, m, t \in \mathbb{N} \wedge n = n? \wedge m = m? \wedge n \leq m) \vee \\ sp\,(sp\,(sp\,(n, m, t \in \mathbb{N} \wedge n = n? \wedge m = m? \wedge n > m,\ \mathtt{t} := \mathtt{n})\,,\ \mathtt{n} := \mathtt{m})\,,\ \mathtt{m} := \mathtt{t}) \end{array} \right) \\
\equiv & \left(\begin{array}{l} (n, m, t \in \mathbb{N} \wedge n = n? \wedge m = m? \wedge n \leq m) \vee \\ sp\,(sp\,(n, m, t \in \mathbb{N} \wedge n = n? \wedge m = m? \wedge n > m \wedge t = n,\ \mathtt{n} := \mathtt{m})\,,\ \mathtt{m} := \mathtt{t}) \end{array} \right) \\
\equiv & \left(\begin{array}{l} (n, m, t \in \mathbb{N} \wedge n = n? \wedge m = m? \wedge n \leq m) \vee \\ sp\,(n?, m, t \in \mathbb{N} \wedge m = m? \wedge n? > m \wedge t = n? \wedge n = m,\ \mathtt{m} := \mathtt{t}) \end{array} \right) \\
\equiv & \left(\begin{array}{l} (n, m, t \in \mathbb{N} \wedge n = n? \wedge m = m? \wedge n \leq m) \vee \\ (n?, m?, t \in \mathbb{N} \wedge n? > m? \wedge t = n? \wedge n = m? \wedge m = t) \end{array} \right)
\end{array}
$$

3.3 Iterative-Form Notation and Convention

A large part of this thesis concerns the calculation of semantic information from loops. Central to the approach taken, is an ability to represent the state changes from one loop iteration to the next. The following notation and convention is used to describe state changes in an iterative form:

- For any variable, v, v_k represents v on the kth iteration of a loop. v_0 represents the value of v on initialisation of a loop.
- For any formula, F, F_k represents F on the kth iteration of a loop. F_0 represents the formula F on initialisation of a loop. $F_{i,k}$ represents F on the kth iteration of a loop executing through path i of the loop body, where there is more than one path through the loop.
- For any function, f, f_k represents the value of f on the kth iteration of a loop. f_0 represents the value of f on initialisation of a loop. $f_{i,k}$ represents the value of f on the kth iteration of a loop executing through path i of the loop body, where there is more than one path through the loop.

3.4 Conclusion

In this chapter much of the foundational work upon which the thesis is developed is presented and discussed. Definitions for functions and notations used in this thesis have been provided. We have provided definitions for the strongest postcondition semantics of imperative programming language constructs. This chapter makes an original contribution in that it describes an algorithm for constructing the strongest postcondition specification from an assignment statement, without relying on the calculation of inverse functions or on the introduction of new variables to the problem, as is the case with previous definitions. We also discuss the problems with existing definitions of strongest postcondition for iterative constructs and procedure calls. These problems are addressed in later chapters.

The strongest postcondition definitions in this chapter for assignment, selection, IO, and declaration have been implemented in a prototype tool[4], as have the substitution and free-variable functions.

This chapter also introduces a notational aid called the *iterative-form* notation. This notation is the foundation for the following chapter in which we present an operational definition for calculating an *iterative-form invariant* for iterative program constructs.

[4]Operational definitions of *sp* for iterative constructs and procedure calls discussed in later chapters have also been partially implemented. The implementations of these definitions are discussed in detail in their respective chapters

Chapter 4

Iterative-Form Semantic Description

"Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by purely deductive reasoning.", C.A.R. Hoare 'An axiomatic basis for computer programming' Comm. ACM 12(10) 576-80, 583 (October 1969)

4.1 The Iterative-Form Invariant

In this chapter, we introduce a special kind of calculable invariant to describe the semantics of an iterating loop. This special invariant, called an *iterative-form invariant*, represents the state of the program variables after an arbitrary $k+1$st iteration in terms of the program variables after an arbitrary kth iteration, where $k \geq 0$. An iterative-form invariant describes the semantic properties yielded as a result of executing a loop under a set of initial conditions, and represents these semantics by relations between kth and $k+1$st iteration instances of variables, where $k \geq 0$. That is, the iterative-form invariant describes recurrence relations or difference equations. For example, the iterative-form equality, $x_{k+1} = x_k + 1$, shows that after a $k+1$st iteration, x will equal what it did after the previous iteration plus 1. In the following chapter we solve all solvable difference equations in the iterative-form invariant yielding a non-iterative invariant, and in chapter 9 verification techniques that use the iterative form invariant are discussed and demonstrated.

Iterative-form invariants can be calculated from loop body statements, by applying the strongest postcondition predicate transformer to the loop body and a formula describing the states after an arbitrary kth iteration. We must provide a description of the kth iteration final state and use this to state the initial conditions on an arbitrary $k+1$st iteration, so that we can apply the strongest postcondition predicate transformer.

The following section discusses the calculation of the initial conditions on an arbitrary $k+1$st iteration. A mechanisable procedure for this calculation is presented which yields these initial conditions directly from a program's code.

4.1.1 Computing Initial Conditions on an Arbitrary Iteration

Considering any loop, `do` $G \rightarrow S$ `od`, executing under the precondition Q. We apply an heuristic algorithm to generate a representation of the initial conditions at the beginning of an arbitrary $k+1$st iteration, where $k \geq 0$. This algorithm defines the weakest condition that must hold in order for the loop to iterate, assuming that all variables used in the loop body and guard are accessible. Using the notation of the previous chapter, v_k represents the instance of a variable v after a kth iteration of a loop. We can use this notational information to state that at the begining of a $k+1$st iteration, any variable v accessible to the loop on the $k+1$st iteration, denoted v_{k+1}, will initially be equal to the value of v_k. These kth iteration values must ensure that the loop guard is *true* in order for the $k+1$st iteration to proceed. Let G_k denote the evaluation of the guard after some kth iteration, then

$$G_k = G\left[v_k/v\right], \text{ for all variables } v \in \mathbf{V}(G).$$

Formally, we denote the weakest conditions required for a $k+1$st iteration by Q_{k+1} and calculate as

$$Q_{k+1} = (k \in \mathbb{N} \wedge G_k \wedge v_{k+1} = v_k \wedge C) \tag{4.1}$$

for all free-variables $v \in \mathbf{V}(Q) \cup \mathbf{V}(S) \cup \mathbf{V}(G)$, and where C denotes the conjunction of all predicates in Q involving only constants or variables not changed by G or S. By definition, a predicate involving only constants or variables not changed by G or S is invariant and must, therefore, hold at the beginning of any arbitrary iteration.

Lemma 4.1. For any loop, $\{Q\}$ `do` $G \rightarrow S$ `od`, Q_{k+1} can always be calculated by (4.1).

Proof. Let v_k represent the value of a variable v after an arbitrary kth iteration of the loop ($k \geq 0$). $\mathbf{V}(G)$ and $\mathbf{V}(Q)$ are finite sets for any program and can always be calculated. Similarly, because S is a finite block of program statements, $\mathbf{V}(S)$, the set of free-variables in S, must also be finite, and therefore calculable. The initial conditions for iterating a loop on an arbitrary $k+1$st iteration, Q_{k+1}, can always be calculated as $\mathbf{V}(G)$, $\mathbf{V}(Q)$, and $\mathbf{V}(S)$ are all bound. $\square$

We introduce algoritm 2 to compute Q_{k+1} satisfying (4.1).

Consider the program segment `i:=1; do` $i \neq 100 \rightarrow$ `i:=i+1; od`, under the precondition *true*, then applying algorithm 2 to obtain Q_{k+1}:

Algorithm 2 Computing Q_{k+1}

For any loop, `do` $G \rightarrow S$ `od`, executing under the precondition Q apply the following steps:

1. Begin with $Q_{k+1} = (k \in \mathbb{N} \wedge G_k)$.
2. For every variable v accessible to the loop, that is, for every variable $v \in (\mathbf{V}(G) \cup \mathbf{V}(S) \cup \mathbf{V}(Q))$, conjoin an equality of the form $v_{k+1} = v_k$ to Q_{k+1}. Every variable will be instantiated either explicitly or by the compiler to a value at all times. At the beginning of a $k+1$st iteration this will be the value that variable was instantiated with after the kth iteration.
3. For all predicates C in Q involving only constants or variables not changed by G or S, conjoin C to Q_{k+1}.

---

1. We begin with
$$Q_{k+1} = (k \in \mathbb{N} \wedge i_k \neq 100)$$
2. The loop precondition is given by $Q = \{i = 1\}$, so $\mathbf{V}(Q) = \mathbf{V}(S) = \mathbf{V}(G) = \{i\}$. Now, i is the only accessible variable of the loop. Conjoining an equality of the form $i_{k+1} = i_k$ to Q_{k+1} gives
$$Q_{k+1} = (k \in \mathbb{N} \wedge i_k \neq 100 \wedge i_{k+1} = i_k)$$
3. There are no equalities in Q involving only constants, so
$$Q_{k+1} = (k \in \mathbb{N} \wedge i_k \neq 100 \wedge i_{k+1} = i_k)$$

These initial conditions are then used as a basis for deriving, through the application of the strongest postcondition predicate transformers, iterative-form difference equations from which invariant properties are determined.

4.1.2 Deriving Iterative-Form Invariants

One very simple but useful way to define the semantics of a looping program is with an iterative-form invariant.

In order to automate our invariant generation method, we need to automatically express the results of loop body statements in an iterative form, $x_{k+1} = f(x_k)$, where f is a function.

The iterative-form invariant holds for any arbitrary $k+1$st iteration where $k \geq 0$.

For any loop $\{Q\}$ `do` $G \rightarrow S$ `od`, we calculate the iterative-form invariant, denoted R_{k+1}, on an arbitrary $k+1$st iteration by

$$R_{k+1} = sp\,(Q_{k+1}, S) \tag{4.2}$$

It can be considered that all statements in S when executed on an arbitrary $k+1$st iteration involve only $k+1$st iteration variable instances. According to the notation presented in the previous chapter, a reference to a variable v in the loop body S on a $k+1$st iteration is denoted v_{k+1}. In order to automate the calculation of (4.2) using the existing definitions for *sp*, we need to actually implement the rule

$$R_{k+1} = sp\left(Q_{k+1}, S\left[v_{k+1}/v\right]\right), \text{ for all variables } v \in \mathbf{V}(S)$$

Example 4.1. Consider the following simple array initialisation program:

```
{a ∈ seq ℕ ∧ N > 0}
i := 0;
do i < N →
    a[i] := i;
    i := i+1;
od
```

The loop precondition is calculated as

$$Q \equiv (a \in \textbf{ seq } \mathbb{N} \wedge N > 0 \wedge i = 0)$$

The initial conditions on an arbitrary $k+1$st iteration, calculated by algorithm 2, are

$$Q_{k+1} \equiv (k \in \mathbb{N} \wedge i_k < N \wedge i_{k+1} = i_k \wedge a_{k+1} = a_k \wedge N > 0)$$

Now, R_{k+1}, the iterative form invariant after the $k+1$st iteration, by the implemented definition of (4.2), is

$$\begin{aligned} R_{k+1} &= sp\left(Q_{k+1}, \ \texttt{a[i]:=i;i:=i+1}\right) \\ &\equiv sp\left(Q_{k+1}, \texttt{a}\left[\texttt{i}_{k+1}\right]_{k+1} := \texttt{i}_{k+1}; \texttt{i}_{k+1} := \texttt{i}_{k+1}+\texttt{1}\right) \\ &\equiv \left(k \in \mathbb{N} \wedge i_k < N \wedge a\left[i_k\right]_{k+1} = i_k \wedge i_{k+1} = i_k + 1 \wedge N > 0\right) \end{aligned}$$

■

The calculation of R_{k+1} is applicable to loops with an arbitrary number of paths through the loop body.

Example 4.2. Consider the following array initialisation program to initialise all even indexed array elements to 2 and all odd indexed array elements to 1:

```
{a ∈ seq ℕ ∧ N > 0}
i := 0;
do i < N →
    if i mod 2 = 0 →
       a[i] := 2;
```

```
        [] i mod 2 ≠ 0 →
            a[i] := 1;
        i := i+1;
    od
```

The loop precondition is calculated as

$$Q \equiv (a \in \texttt{seq}\,\mathbb{N} \wedge N > 0 \wedge i = 0)$$

The initial conditions on an arbitrary $k+1$st iteration, calculated by algorithm 2, are

$$Q_{k+1} \equiv (k \in \mathbb{N} \wedge i_k < N \wedge i_{k+1} = i_k \wedge a_{k+1} = a_k \wedge N > 0)$$

Now, R_{k+1}, the iterative form invariant after the $k+1$st iteration, by (4.2), is

$$R_{k+1} \quad \equiv \quad \left(\begin{array}{l} k \in \mathbb{N} \wedge i_k < N \wedge i_{k+1} = i_k + 1 \wedge N > 0 \wedge \\ \left(\begin{array}{l} \left(i_k \bmod 2 = 0 \wedge a\left[i_k\right]_{k+1} = 2\right) \vee \\ \left(i_k \bmod 2 \neq 0 \wedge a\left[i_k\right]_{k+1} = 1\right) \end{array} \right) \end{array} \right)$$

■

Lemma 4.2. For any loop, `do` $G \rightarrow S$ `od`, where S contains no iterative structures, R_{k+1} is always able to be calculated.

Proof. If S contains no iterative structures then S must only contain sequences or selections of assignment statements and procedure calls (disregarding non-executable statements and aborts). Therefore, if Q_{k+1} can always be calculated, then $sp\,(Q_{k+1}, S)$ is a fixed depth calculation by definition of the strongest postcondition predicate transformers. By Lemma 4.1, Q_{k+1} can always be calculated, so, R_{k+1} can always be calculated. □

Lemma 4.3. For any loop, `do` $G \rightarrow S$ `od`, where S contains no iterative structures, the number of disjuncts in R_{k+1} is equal to the number of paths through S.

Proof. This result is obtainable directly from the definitions of the strongest postcondition predicate transformer for assignment and selection by induction on program structure. □

4.1.3 Verification using the Iterative-form Invariant

There are a number of potential uses for the iterative form invariant in verification and defect detection.

With respect to formal proof of correcness, the iterative form-invariant for a loop can be used in a proof by mathematical induction to prove that the loop maintains a given invariant. This method of proof is discussed in chapter 9. We show the method to be complete for proving partial correctness with respect to the fundamental invariance theorem [47].

In some cases a program's specification is given as a function. In order to prove that a program is correct with repsect to this function we need to show that the program syntax implements a semantically equivalent function to the one specified. For example, a specification for a function `fn` to return twice the square of a natural number might be presented as

$$\begin{aligned}&\mathit{fn} : \mathbb{N} \to \mathbb{N}\\ &\mathit{fn}(a) = 2a^2\end{aligned}$$

Consider the following program to implement this function:

```
function fn(a:  IN ℕ) :  ℕ {
    x := a; x := x*x; x := 2*x; return x
}
```

In order to prove correctness with respect to the specification, we need to show that $x = 2a^2$ when it is returned. Applying the *sp* predicate transformer to the program, we calculate the following strongest postcondition prior to the return statement:

$$sp\,(a \in \mathbb{N}, \texttt{x:=a; x:=x*x; x:=2*x;}) \equiv \left(a \in \mathbb{N} \wedge x = 2a^2\right)$$

The derived semantics are proof that the program is correct with repect to its specification.

This technique can be extended to verify iterative programs for which the specification is given as an iterative-form function or iterative-algorithm.

Example 4.3. For example, consider the following program that implements the iterative Newton-Raphson method to calculate the square root of a positive number to within a given tolerance.

```
{x, nr, tolerance ∈ ℝ ∧ nr > 0}
y := nr/2; x := (y + nr/y)/2;
do (|y − x| ≥ tolerance)→
    y := x; x := (y + nr/y)/2;
od;
```

R_{k+1} is calculated by (4.2) as

$$R_{k+1} \equiv \left(\begin{array}{l} k \in \mathbf{N} \wedge nr_{k+1} = nr_k \wedge nr_{k+1} > 0 \wedge r_{k+1} = r_k \wedge \\ tolerance_{k+1} = tolerance_k \wedge |y_k - x_k| \geq tolerance_k \wedge \\ y_{k+1} = x_k \wedge x_{k+1} = \frac{y_{k+1} + (nr/y_{k+1})}{2} \end{array}\right)$$

The Newton-Raphson method to approximate the zeros of an equation $f(x) = 0$ involves iteratively calculating:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

We reason that the loop above implements the Newton-Raphson method for approximating the square root of a number *nr*, that is, it approximates the zero of $f(x) = x^2 - nr$. Let x_k be the successive approximations obtained through application of the Newton-Raphson method. We have

$$\begin{aligned}
x_{k+1} &= x_k - \frac{f(x_k)}{f'(x_k)} \\
&= x_k - \frac{{x_k}^2 - nr}{2x_k} \\
&= \frac{2{x_k}^2 - \left({x_k}^2 - nr\right)}{2x_k} \\
&= \frac{{x_k}^2 + nr}{2x_k} \\
&= \frac{x_k + \frac{nr}{x_k}}{2}
\end{aligned}$$

By inspection of the iterative invariant R_{k+1}, we see that

$$\begin{aligned}
R_{k+1} &\Rightarrow x_{k+1} = \frac{y_{k+1} + (nr/y_{k+1})}{2} \wedge y_{k+1} = x_k \\
&\Rightarrow x_{k+1} = \frac{x_k + (nr/x_k)}{2}
\end{aligned}$$

Therefore, the above implementation is at least a partially correct implementation of the Newton-Raphson method for approximating the square root of a number, nr. ∎

The iterative-form invariant can also be manipulated in a number of ways to assist reviewers in reasoning about correctness. The next section introduces some simple manipulations that prove useful in inspection.

4.1.4 Manipulations on the Iterative Form

The iterative-form invariant derived from a program is often a very large predicate. The usefulness of such a predicate is questionable. In many cases, the whole predicate is not necessary for verifying certain properties. For example, in the Newton method program above, one only needed to consider the relationship between x and nr to determine the correctness of the implemented algorithm. In many cases, focussing on the relationships among a small number of program variables is all that is required to reason about many properties. In focussing on a subset, S, of the program variables, substitutions implied by

R_{k+1} should be made so that as much information as possible is presented about the program in terms of the S variables.

An iterative-form invariant, focussed on a subset, S, of the program variables is often best presented as a table (or map) where each column represents one of the variables of S, and each row represents the relationship among the variables of S implied by a disjunct of R_{k+1}. Each row should be qualified by the conditions, yielding that row. These conditions should correspond to the terms in the respective disjuncts, yielded as a result of the guarded commands in the loop body. In summary, algorithm 3 is suggested for generating a variable map from an iterative-form invariant.

Algorithm 3 Computing a variable map from R_{k+1}

For any iterative-form invariant R_{k+1} and given set of focus variables $S \subseteq \mathbf{V}(R_{k+1})$, apply the following steps:

1. Create a column for every variable $v \in S$. The column heading is v_{k+1}
2. Create an additional column called conditions
3. For every disjunct $R_{i,k+1}$ in R_{k+1}, create a row as follows:
   - For every variable $v \in S$ copy all terms of the form $v_{k+1} = exp$ into the column for v.
   - Write the conjunction of the terms C in $R_{i,k+1}$ which were yielded from a guard into the conditions column for that row.

The relationship between *input* and *output* is a special case of the above. The input and output streams are represented by the special variables, Ψ and Ω respectively. The correctness of many programs that manipulate an input to produce and output, can be reasoned about in terms of the changes to Ψ and Ω only. A common specification technique for such programs is the use of an input-output map, or diagram. The application of algorithm 3 with Ψ and Ω yields the derived input-output map for verification.

Consider the text reformatting algorithm of Program 1.

It is intended that this program read from a text file and reformat the output in accordance with the specification in figure 4.1.

Due to the lack of formality in the specification and the complexity of the code, it is not possible to prove that Program 1 is correct with respect to the given specification, without first formalising the specification. It is extremely difficult, if not impossible, to automate such a proof due to difficulty in automatically identifying a suitable invariant.

Given that formal proof of correctness is probably not feasible in this situation, we consider an inspection based on human reasoning. Employing an inspection approach focussing on semi-formal reasoning has the potential to provide identification of a number of defects in this program as part of an argument of

Program 1 The Naur text reformatting example

```
MAXPOS := 31;
more := 1;
bp := 0;
fill := 0;
do (more = 1) →
    read c;
    if c=BL ∨ c=LF ∨ c=EOT →
        if c=EOT →
            more := 0;
        fi;
        if fill+bp+1≤MAXPOS →
            write BL; fill:=fill+1;
        [] fill+bp+1>MAXPOS →
            write LF; fill:=0;
        fi;
        write buf[1..bp];
        fill:=fill+bp;
        bp:=0;
    [] c≠BL ∧ c≠LF ∧ c≠EOT →
        if bp=MAXPOS →
            more:=0;
            write 'Line Too Long';
        [] bp≠MAXPOS →
            bp:=bp+1;
            buf[bp]:=c;
        fi;
    fi;
od;
```

correctness. Such reasoning is made more repeatable by mechanically extracting semantic information from the code.

Applying the technique described above for calculating the iterative-form invariant, we can derive R_{k+1} for the main loop of this program.

The derived invariant (Figure 4.2) is large, and hence, quite difficult to reason about. We apply 3 with Ψ and Ω to present only the parts of this invariant which deal with input and output. The input and output variables, Ψ and Ω, were chosen in this case because the specification is written in terms of mappings from inputs to output. The derived mappings, when inspected, allow a reader to identify a number of defects. These mappings can be described by

1. Each output line shall consist of 30 characters.
2. Each output line shall contain as many whole words as possible
3. No word shall be split across lines in the output
4. Words in the input shall be seperated by one or more blanks (BL) or line feed (LF) characters. Words in the output shall be seperated by only one blank and each line is ended with a linefeed character.
5. An error message shall be output if a word is more than 30 characters long.
6. A file shall be terminated by the EOT character

Figure 4.1: Specification for the text reformatting example in Program 1.

$$\left(\begin{array}{l}
k \in \mathbb{N} \wedge \\
\left(\begin{array}{l}
\Psi_{k+1} = \Psi_k + EOT \wedge \\
more_{k+1} = 0 \wedge c_{k+1} = EOT \wedge fill_{k+1} = fill_k + 1 + bp_k \wedge \\
bp_{k+1} = 0 \wedge buf_{k+1} = buf_k \wedge MAXPOS = 31 \wedge fill_k + bp_k + 1 \leq MAXPOS \wedge \\
\Omega_{k+1} = \Omega_k + BL + buf[1..bp_k]
\end{array}\right) \vee \\
\left(\begin{array}{l}
\Psi_{k+1} = \Psi_k + c_{k+1} \wedge \\
more_{k+1} = 1 \wedge c_{k+1} \in \{BL, LF\} \wedge fill_{k+1} = fill_k + 1 + bp_k \wedge \\
bp_{k+1} = 0 \wedge buf_{k+1} = buf_k \wedge MAXPOS = 31 \wedge fill_k + bp_k + 1 \leq MAXPOS \wedge \\
\Omega_{k+1} = \Omega_k + BL + buf[1..bp_k]
\end{array}\right) \vee \\
\left(\begin{array}{l}
\Psi_{k+1} = \Psi_k + EOT \wedge \\
more_{k+1} = 0 \wedge c_{k+1} = EOT \wedge fill_{k+1} = bp_k \wedge \\
bp_{k+1} = 0 \wedge buf_{k+1} = buf_k \wedge MAXPOS = 31 \wedge fill_k + bp_k + 1 > MAXPOS \wedge \\
\Omega_{k+1} = \Omega_k + LF + buf[1..bp_k]
\end{array}\right) \vee \\
\left(\begin{array}{l}
\Psi_{k+1} = \Psi_k + c_{k+1} \wedge \\
more_{k+1} = 1 \wedge c_{k+1} \in \{BL, LF\} \wedge fill_{k+1} = bp_k \wedge \\
bp_{k+1} = 0 \wedge buf_{k+1} = buf_k \wedge MAXPOS = 31 \wedge fill_k + bp_k + 1 > MAXPOS \wedge \\
\Omega_{k+1} = \Omega_k + LF + buf[1..bp_k]
\end{array}\right) \vee \\
\left(\begin{array}{l}
\Psi_{k+1} = \Psi_k + c_{k+1} \wedge \\
more_{k+1} = 0 \wedge c_{k+1} \notin \{BL, LF, EOT\} \wedge fill_{k+1} = fill_k \wedge \\
bp_{k+1} = MAXPOS \wedge buf_{k+1} = buf_k \wedge MAXPOS = 31 \wedge \\
\Omega_{k+1} = \Omega_k +' \text{Line Too Long}'
\end{array}\right) \vee \\
\left(\begin{array}{l}
\Psi_{k+1} = \Psi_k + c_{k+1} \wedge \\
more_{k+1} = 1 \wedge c_{k+1} \notin \{BL, LF, EOT\} \wedge fill_{k+1} = fill_k \wedge \\
bp_{k+1} = bp_k + 1 \wedge bp_k \neq MAXPOS \wedge buf_{k+1}[bp_{k+1}] = c_{k+1} \wedge MAXPOS = 31 \wedge \\
\forall i \in [1..bp_k]\,(i \neq bp_{k+1} \Rightarrow buf_{k+1}[i] = buf_k[i]) \wedge \\
\Omega_{k+1} = \Omega_k
\end{array}\right)
\end{array}\right)$$

Figure 4.2: The derived iterative-form invariant from the text reformatting program main loop.

Row	**Ψ (Input)**	**Ω (Output)**	**Conditions**
1	*EOT*	$BL + buf\,[1..bp_k]$	$fill_k + bp_k + 1 \leq MAXPOS$
2	*BL*	$BL + buf\,[1..bp_k]$	$fill_k + bp_k + 1 \leq MAXPOS$
3	*LF*	$BL + buf\,[1..bp_k]$	$fill_k + bp_k + 1 \leq MAXPOS$
4	*EOT*	$LF + buf\,[1..bp_k]$	$fill_k + bp_k + 1 > MAXPOS$
5	*BL*	$LF + buf\,[1..bp_k]$	$fill_k + bp_k + 1 > MAXPOS$
6	*LF*	$LF + buf\,[1..bp_k]$	$fill_k + bp_k + 1 > MAXPOS$
7	*PC*	*Line Too Long*	$bp_k = MAXPOS$
8	*PC*	*	$bp_k \neq MAXPOS$

*When $bp_k \neq MAXPOS$, no output is generated and $buf\,[bp_k + 1] = c_{k+1}$ is maintained. That is, the character read is written to the word buffer, buf. PC is a generated definition to represent a character that is not a `BL`, `LF`, or `EOT` . It is identifiable from the branch guards.

We can identify the following defects, simply by reasoning about the derived semantic information with respect to the given specification. Most of these defects can be found by analysing the mapping alone.

1. Every line where the first word is less than MAXPOS characters long begins with a BL[1]. This is evident from rows 1,2, and 3 of the above table.

2. Redundant BLs are not removed unless they will be the MAXPOS+1st character on a line. This is evident from rows 2 and 5 of the above table.

3. Multiple LFs are replaced by redundant BLs unless they will be the MAXPOS+1st character on a line. This is evident from rows 3 and 6 of the above table.

4. If the first word on a line is MAXPOS characters long, then a redundant LF is inserted before the word. This can be reasoned by examining rows 4, 5 and 6 of the above table. If a word has just been read that is MAXPOS characters long and it is followed on the $k+1$st iteration by either a BL, LF, or EOT, then $fill_k + bp_k + 1$ must be greater than MAXPOS because $bp_k = MAXPOS$. In these cases a LF is output before the word (contents of buf).

5. An EOT character is replaced by a BL unless it is the last character on the line when it is replaced by a LF. This is evident from rows 1 and 4 of the above table.

6. If a word is greater than MAXPOS characters long, then the error message may be printed past the allowable last position as there is no linefeed output before the error message. As an example, consider that MAXPOS-1 characters have been written to a line prior to reading the long word. When, on some arbitrary $k+1$st iteration, bp is greater than MAXPOS, row 7 of the above table indicates that the error message *Line Too Long* is output. This will mean that more than the allowable number of characters will be output to the line

7. MAXPOS is set to 31. Lines will be printed up to MAXPOS characters and words may contain up to MAXPOS characters. MAXPOS should be 30, according to the given specification.

The defects identified were found through reasoning about the semantic information generated from the code, not by inspecting the code itself. There was

[1]It is evident from the full derived iterative invariant that $fill_k + bp_k + 1$ is the count of the number of characters read since the last LF was output.

no requirement to construct test cases or cases for symbolic execution of the code. The reasoning process employed involved no formal rules of proof. The derivation of a similar table from two loop iterations is automated in an early version of the prototype tool. Output from this prototype is presented in Fig. 4.3.

DEFINITIONS:

LF ::= "$"
EOT ::= "/"
MAXPOS ::= 31
BL ::= " "
_Def0 ::= ¬EOT AND ¬LF AND ¬BL

IO MAPPING

Input	Output	
c_1.BL	BL.c_1	●
BL.BL	BL.BL	●
LF.BL	BL.BL	●
c_1.LF	BL.c_1	●
BL.LF	BL.BL	●
LF.LF	BL.BL	●
c_1.EOT	BL.c_1	●
BL.EOT	BL.BL	●
LF.EOT	BL.BL	●
BL.c	BL	●
LF.c	BL	●
c_1.c		●
EOT	BL	●

Figure 4.3: Screenshot showing the IO mappings after 2 iterations, from the Naur text reformatting example. 4

4.2 Conclusion

In this chapter we introduce an operational definition for constructing an iterative-form invariant from iterative program constructs. An iterative-form invariant describes the iterative relationships that hold among variables in progressing from an arbitrary kth iteration to a $k+1$st iteration. The iterative-form invariant is calculated by application of Algorithm 2 and by equation (4.2). Algorithm 2 and equation (4.2) have been implemented in the prototype tool.

We demonstrate that an iterative invariant can be used to reason about correctness and is particularly useful in reasoning about correctness of a program with respect to an iterative, functional specification. As a consequence of Lemma 4.3, one weakness of using the iterative-form invariant for assisting verification-based inspection is that a loop with a large number of paths through the loop body will have a large iterative-form invariant. We address this issue by defining Algorithm 3 to allow a user of the specification to focus on relationships among a subset of program variables. This 'slicing' technique is demonstrated with a text reformatting example. Algorithm 3 has been implemented in an older version the prototype only for the special case of focussing on the input (Ψ) and output (Ω). The output from this version of the tool is presented in Fig. 4.3. This functionality is not fully implemented in the current version of the prototype. It is envisaged that in this and later versions of the tool, a user will be able to dynamically select variables to focus the specification. Allowing different focusses to be investigated during inspection would likely be a useful static analysis technique.

The iterative-form invariant as defined in this chapter forms the basis for the calculation of non-iterative invariants in the following two chapters.

Chapter 5

Non-Branching Loop Semantics

Formal, semantic description of software has the potential to play a central role in the verification, evolution, and reuse of software components. Of the information that makes up this semantic description, loop invariants and postconditions have been notoriously difficult to provide. Present techniques for providing invariants can be classified as either inference approaches or heuristic and algorithmic approaches. Inference approaches [30] use output assertions to infer candidate invariants. These approaches are generally reliant on human specification of appropriate initial conditions and execution suites in order to generate the output from which candidate invariants are inferred. They are, therefore, not fully-automatable and suffer from the same effort and assuredness problems as does software testing. Algorithmic approaches [41, 62, 80, 9, 11] define mechanical processes to generate invariants for simple classes of loops by formal calculation. Despite the repeatability of algorithmic techniques, none have been used to implement fully-automated analysis systems [36]. This is due to the complexity and enormous computational effort required to apply these techniques to derive strong enough assertions to be used in formal proofs [36]. The majority of algorithmic approaches rely on heuristics to infer invariants for complex classes of loops.

An alternative approach is to weaken the requirement of deriving invariants and postconditions strong enough to support formal proof, to the derivation of semantic information, including invariants, which are strong enough to support human, reasoning-based code reading techniques, such as *reading by stepwise abstraction*. To this end, we describe techniques to derive invariant and postcondition information from code syntax. The iterative-form invariant of the previous chapter is an integral part of this information. The approach taken in this chapter involves the attempted solution of difference equations in the iterative-form invariant and induction on guards to yield strong non-iterative invariants. As such, it is an algorithmic approach similar to that described in [62]. The approach presented here differs from that of [62] in that the iterative forms of the difference equations are generated from the code through the application of the strongest postcondition predicate transformer. This generation of iterative-form invariants was described in the previous chapter. No "built-in" invariants are assumed, and the iterative count is used to derive ranges on invariant quantified formulae.

This chapter describes a repeatable procedure, and its accompanying theoretical basis, to derive loop invariants and postconditions from the code syntax of loops with non-branching loop bodies. We also describe certain calculations, performed as part of the derivation process, that support boundary value defect detection. Inspection rules are introduced which provide repeatable guidelines for the use of the derived information in finding defects and reasoning about correctness. The goal of this approach is to automate the generation of invariant assertions and postconditions to provide inspectors with a basis for reasoning about a program's correctness. The next chapter extends the theory presented here to provide a procedure that works to derive invariants and postconditions for certain classes of loops with multi-path loop bodies. The derivation procedure and its theory are presented with a number of examples.

5.1 A Formal Basis for Calculation[1]

In order to automate the derivation of invariants and postconditions from code, a mechanical technique for deriving semantics from individual statements is needed. To provide the required mapping from syntax to semantics, the predicate calculus *strongest postcondition predicate transformer* is used.

Unlike inferring invariants from program traces, as described in [30], the derivation approach can be applied without any consideration to test, or execution, suites. In effect, this approach exercises a program under all of the possible inputs described by a given precondition and describes all of the achievable states under those preconditions.

Chapter 3 introduced and extended the basic strongest postcondition predicate transformers and provided examples of their use. It identified the problem of directly calculating the strongest postcondition of an iterating segment of code in which the calculation depth is not fixed. In this chapter, we make use of the relationship between loops and induction to derive the invariant properties of code under any state allowed by the precondition of that code. Of central importance to this calculation, is a framework for describing the functions by which variables change with successive iterations of a loop body. When considering loops with non-branching bodies, we apply difference equation, often referred to as recurrence relation, theory to describe, necessarily discrete, variable changes. By solving the difference equations implicit in the iterative-form invariant, and substituting these solutions we arrive at an invariant describing the relationships between variables and the iteration count. The extraction of the iterative-form invariant and the solutions to many common difference

[1] Most of the required predicate calculus theory was presented in the previous chapter. Further formal concepts and notations are introduced as required.

equations have been mechanised in a prototype tool which forms part of this submission. Therefore, in many cases, generation of an invariant from non-branching loops is completely automated.

5.1.1 Loop Bodies as Systems of Difference Equations

In this section we present mathematical techniques capable of describing the dynamics of variable state information. In order to derive an invariant we need to be able to describe how a variable, and any relationships involving that variable, change from one discrete unit of time to the next. In the case of loops, these discrete units of time are iterations. The values over time (iterations) that a variable v is assigned can be described by the sequence $v_0, v_1, ..., v_k, v_{k+1}, ..., v_{n-1}, v_n$, where n is a finite natural number[2].

Let us now consider the very simple case where the body of the loop increments a variable v by 1. That is, the loop body contains the assignment, `v:=v+1`. Given that this is the only statement in the body of the loop to update v, we can say that $v_{k+1} = v_k + 1$, for any $k \geq 0$. Assuming that v is assigned 0 as part of the initialisation, we can show that:

$$\begin{aligned}
v_0 &= 0 \\
v_1 &= v_0 + 1 = 0 + 1 = 1 \\
v_2 &= v_0 + 1 + 1 = 0 + 1 + 1 = 2 \\
v_k &= v_0 + 1 + ...to\, k\, terms... + 1 \\
&= v_0 + k \\
&= k \\
v_{k+1} &= v_k + 1 = k + 1
\end{aligned}$$

In doing this, we define the value for a variable after any loop iteration in terms of the initialisation of the variable and the iteration count. If a variable is not initialised prior to use on the right hand side of a self assignment in the body of a loop, then the above calculation can only be performed with respect to some arbitrary initial value of the variable. For an uninitialised variable, v, we use v_0 to denote this arbitrary initial value. Importantly, we have identified at least a quality defect and at worst a non-deterministic operation, in the program.

Definition 4. An equation which expresses a value of a term in a sequence as a function of other terms in the sequence is called a *difference equation* or *recurrence relation* in iterative form. An equation which expresses the value of a term v_{k+1} of the sequence $\langle v_n \rangle$ in terms of the value of the term v_k is called a *first-order difference equation*. If for a given difference equation, we can determine an explicit formula for v_k, where $k \geq 0$, in terms of v_0, then we

[2]Note that by describing this sequence as finite, we are assuming that the loop terminates. That is, in order to derive an invariant to be used in verifying the partial correctness of a loop, we ignore the question of termination. Termination is investigated independently in chapter 6.

can solve the difference equation. If such a formula exists, then we will call it the *solution* to the difference equation and denote it $v(k)$.

For simplicity, this thesis deals only with first-order difference equations, and examples that manipulate integers and sequences of integers only. However, the method presented in this chapter is applicable to any solvable difference equation. It is important to note that, when dealing with loops with non-branching bodies, all iterative changes in scalar variables are described by first-order difference equations. That is, for some variable v on some iteration $k+1$, we only have access to v_k, because if the body of the loop updates v, then v_i, where $i < k$ no longer exists on the $k+1$st iteration. If the body of the loop does not update v on a $k+1$st iteration, then $v_{k+1} = v_k$, by definition of R_{k+1}. Therefore, we cannot have an equality of the form $v_{k+1} = f(v_i)$, where $i < k$.

In order to apply an invariant generation method based on solving difference equations, we need to automatically express the results of executing an arbitrary loop body iteration in terms of iterative form equalities. The iterative-form invariant, calculated by the method described in the previous chapter, provides these iterative-form equalities.

Consider the exponentiation example in Program 2. This code is used as an example in [58] to demonstrate a method of invariant generation from failed proof attempts. We need to verify that under the precondition $(X, Y \in \mathbb{N} \wedge x = X \wedge y = Y)$, the program terminates and establishes $r = X^Y$. The loop's

Program 2 Exponentiation calculation example[58]

$\{X, Y \in \mathbb{N} \wedge x = X \wedge y = Y\}$

```
r := 1;
do y > 0 →
    r := r*x;
    y := y-1;
od
```

precondition is

$$Q \equiv sp(X, Y \in \mathbb{N} \wedge x = X \wedge y = Y,\ \mathrm{r := 1}) \equiv (r = 1 \wedge X, Y \in \mathbb{N} \wedge x = X \wedge y = Y)$$

The initial conditions on an arbitrary $k+1$st iteration are

$$Q_{k+1} \equiv (k \in \mathbb{N} \wedge r_{k+1} = r_k \wedge x_{k+1} = x_k \wedge y_{k+1} = y_k \wedge y_k > 0)$$

Now, R_{k+1}, the iterative form invariant after the $k+1$st iteration, is given by

$$\begin{aligned} R_{k+1} &= sp\left(Q_{k+1},\ r := r * x; y := y - 1\right) \\ &\equiv (k \in \mathbb{N} \wedge r_{k+1} = r_k x_k \wedge x_{k+1} = x_k \wedge y_{k+1} = y_k - 1 \wedge y_k > 0) \end{aligned}$$

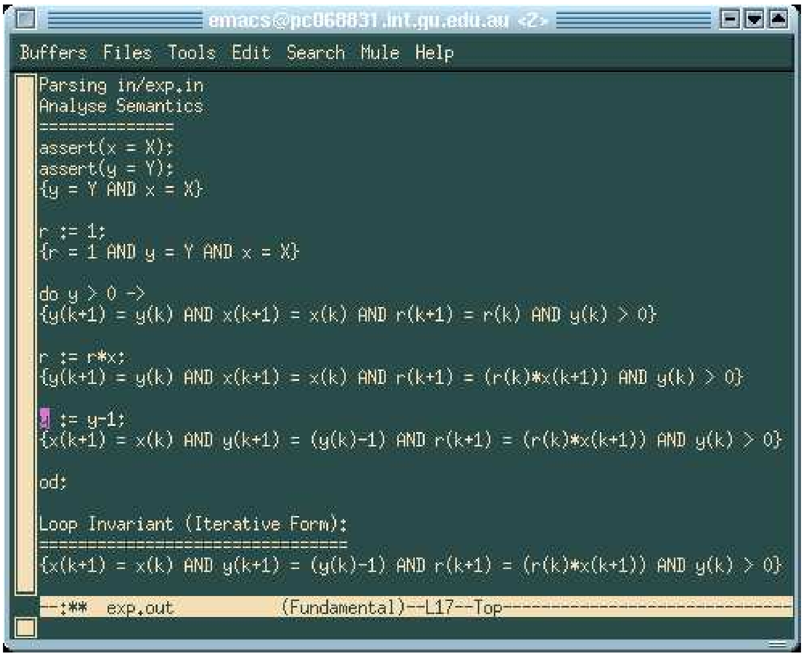


Figure 5.1: Intermediate output file from prototype tool

As mentioned earlier, the derivation of this iterative invariant has been automated. Figure 5.1 shows one of the intermediate files generated by the current version of the prototype tool.

R_{k+1} describes the invariant properties of a loop (not bypassed) in iterative form. In many cases, the iterative form invariant is less useful for verification than a form where the iterative form difference equations are solved[3]. Solving the iterative form difference equations in R_{k+1} results in a non-iterative form invariant which is more useful for verification. This example is continued in this light in section 5.2. Next we present the solutions to some commonly occurring difference equations.

5.1.2 Common Difference Equation Solutions Derived

This section presents a derivation of the solutions to a number of iterative form difference equations that occur frequently in programs. These solutions have been implemented in a prototype tool (See screenshots in Appendix B).

Solution to $x_{k+1} = \alpha x_k + \beta$

Often assignments in loops yield difference equations of the form $x_{k+1} = \alpha x_k + \beta$, where α and β are constants. Such equations are often referred to as *first-order*

[3]If an invariant was given then the iterative-form invariant is all that is required to demonstrate partial correctness (see chapter 8). In this example, there is no invariant given.

linear difference equations. A solution to first-order linear difference equations is derived below:

$$
\begin{aligned}
x_{k+1} &= \alpha x_k + \beta \\
&= \alpha\left(\alpha x_{k-1} + \beta\right) + \beta \\
&= \alpha^2\left(\alpha x_{k-2} + \beta\right) + \beta\left(\alpha + 1\right) \\
&= \alpha^3 x_{k-2} + \beta\left(\alpha^2 + \alpha + 1\right) \\
&\ . \\
&\ . \\
&\ . \\
&= \alpha^{k+1} x_0 + \beta\left(\alpha^k + \alpha^{k-1} + ... + \alpha^2 + \alpha + 1\right)
\end{aligned}
$$

Note that when $\alpha = 1$, we get, $x_{k+1} = x_0 + \beta\left(k+1\right)$. So the solution to the difference equation $x_{k+1} = x_k + \beta$ where $\alpha = 1$ is given by:

$$x(k) = x_0 + \beta k \tag{5.1}$$

It can be proved that $\left(\alpha^k + \alpha^{k-1} + ... + \alpha^2 + \alpha + 1\right) = \left(\frac{1-\alpha^{k+1}}{1-\alpha}\right)$, where $k \geq 0$ and $\alpha \neq 1$.

Hence, the solution of the difference equation $x_{k+1} = \alpha x_k + \beta$, when $\alpha \neq 1$ and $k \geq 0$, is:

$$x(k) = \alpha^k x_0 + \beta\left(\frac{1-\alpha^k}{1-\alpha}\right) \tag{5.2}$$

Consider the following array initialisation example:

```
i := 0;
do i < N →

    a[i] := 0;
    i := i + 1;

od
```

R_{k+1} describes the difference equation $i_{k+1} = i_k + 1$. We apply equation (5.1) to solve this difference equation and express the value of i on some $k+1$st iteration of the loop in terms of the initialisation of i and the iteration count:

$$
\begin{aligned}
i_{k+1} &= i_k + 1 \\
&= i(k+1) \\
&= i_0 + 1.(k+1) \\
&= 0 + k + 1 \\
&= k + 1
\end{aligned}
$$

Solution to $x_{k+1} = (x_k)^n$

Occasionally an assignment statement in the body of a loop will look like, `x:=x`^{n}, yielding the iterative-form difference equation, $x_{k+1} = (x_k)^n$. The solution to the iterative equation $x_{k+1} = (x_k)^n$, is given without proof by (5.3).

$$x(k) = (x_0)^{\left(n^{k+1}\right)} \tag{5.3}$$

Using equations (5.3) and (5.1) we can determine invariants for loops such as the following:

```
i:=0;
x:=5;
do i != 3 ->
     x := x^5;
     i := i + 1;
od
```

We know $i_0 = 0$ and $x_0 = 5$, and we can say for any arbitrary $k \geq 0$, $i_{k+1} = k+1$ by (5.1), and $x_{k+1} = (x_0)^{\left(5^{k+1}\right)} = 5^{\left(5^{k+1}\right)}$ by (5.3). For example after the second iteration, $i_2 = i(2) = 2$ and $x_2 = x(2) = 5^{\left(5^2\right)} = 5^{(25)}$. By substitution, we identify the invariant relationship $x = 5^{\left(5^i\right)}$.

Solutions to sums of sequences

Often loops are used to find the sum or product of a sequence of values. We can use difference equations to derive the invariant properties of these types of loops also.

Case 1: Variable Summed Prior to Changing

Consider the following loop to sum even numbers:

```
i := 0; sum := 0;
do i <= N ->
     sum := sum + i;
     i := i + 2;
od;
```

We know that $i_{k+1} = (i_k + 2) = 2(k+1)$, by (5.1). In order to calculate an invariant, we also need to be able to describe the value of sum_{k+1}. This is not so easy, as the assignment yields the iterative form difference equation, $sum_{k+1} = sum_k + i_k$. The problem is that i_k can't be considered the same as β in equation (5.1), because i_k is not constant. Instead we need a solution to iterative-form difference equations of the form $x_{k+1} = \alpha x_k + \beta i_k$, where i_k is also a variable changed in the body of the loop. This solution is derived below:

$$\begin{aligned}
x_{k+1} &= \alpha x_k + \beta i_k \\
&= \alpha\left(\alpha x_{k-1} + \beta i_{k-1}\right) + \beta i_k \\
&= \alpha^2 x_{k-1} + \alpha\beta i_{k-1} + \beta i_k \\
&= \alpha^2\left(\alpha x_{k-2} + \beta i_{k-2}\right) + \alpha\beta i_{k-1} + \beta i_k \\
&= \alpha^3 x_{k-2} + \alpha^2\beta i_{k-2} + \alpha\beta i_{k-1} + \beta i_k \\
&= \alpha^3 x_{k-2} + \beta\left(\alpha^2 i_{k-2} + \alpha i_{k-1} + i_k\right) \\
&. \\
&. \\
&. \\
&= \alpha^{k+1} x_0 + \beta\left(\alpha^k i_0 + \alpha^{k-1} i_1 + ... + \alpha^1 i_{k-1} + i_k\right) \\
&= \alpha^{k+1} x_0 + \beta\left(\sum_{j=0}^{k} \alpha^j i\left(k-j\right)\right)
\end{aligned}$$

So, the solution to an equation of the form $x_{k+1} = \alpha x_k + \beta i_k$ is

$$x(k) = \alpha^k x_0 + \beta\left(\sum_{j=0}^{k-1} \alpha^j i\left(k-1-j\right)\right) \tag{5.4}$$

A common special case occurs when α and β equal 1. In these cases the iterative form will be $x_{k+1} = x_k + i_k$. This form of equation has the solution

$$x(k) = x_0 + \left(\sum_{j=0}^{k-1} i\left(j\right)\right) \tag{5.5}$$

Note that these solutions involve summing over the solution to the difference equation describing i.

We can now calculate strong invariant properties for the loop above using (5.1) and (5.5). By (5.1) $i(k) = 2k$, so the value of sum after the $k+1$st iteration, sum_{k+1}, can be calculated as:

$$\begin{aligned}
sum_{k+1} &= sum(k+1) \\
&= sum_0 + 1.\left(\textstyle\sum_{j=0}^{k} i(j)\right) \\
&= 0 + \left(\textstyle\sum_{j=0}^{k} i(j)\right) \\
&= \textstyle\sum_{j=0}^{k} i\left(j\right) \\
&= \textstyle\sum_{j=0}^{k} 2j
\end{aligned}$$

The prototype tool generates equivalent results for this example (see Fig. **??**).

Case 2: Variable Summed After Changing

We can use equations (5.4) and (5.5) if i is added to x before i is updated in the loop body. If i is updated before being added to x, then the iterative form $x_{k+1} = \alpha x_k + \beta i_{k+1}$ is yielded by the strongest postcondition calculation. A solution for this iterative form is derived below.

$$\begin{aligned}
x_{k+1} &= \alpha x_k + \beta i_{k+1} \\
&= \alpha^{k+1} x_0 + \beta\left(\alpha^k i_1 + \alpha^{k-1} i_2 + ... + \alpha^1 i_k + i_{k+1}\right) \\
&= \alpha^{k+1} x_0 + \beta\left(\sum_{j=0}^{k} \alpha^j i\,(k+1-j)\right) \\
\text{So, } x(k) &= \alpha^k x_0 + \beta\left(\sum_{j=0}^{k-1} \alpha^j i\,(k-j)\right) \qquad (5.6)
\end{aligned}$$

When α and β equal 1, the solution is:

$$x(k) = x_0 + \left(\sum_{j=1}^{k+1} i\,(j)\right) \tag{5.7}$$

Sums of indexed data structures representing sequences

Often the sequences we are summing over are indexed data structures such as arrays rather than scalar variables such as i in the above examples. For example, consider an assignment yielding, $x_{k+1} = \alpha.x_k + \beta.a\left[e\left(i_k\right)\right]$ or $x_{k+1} = \alpha.x_k + \beta.a\left[e\left(i_{k+1}\right)\right]$, where i is a variable that is also changing in the body of the loop and e is a function yielding natural numbers (possibly the identity function).

For these cases the rules for solving the resulting difference equations are:

1. For an iterative form $x_{k+1} = \alpha.x_k + \beta.a\left[e\left(i_k\right)\right]$ the solution is:

$$x(k) = \alpha^k x_0 + \beta\left(\sum_{j=0}^{k-1} \alpha^j a\left[e\left(i(k-1-j)\right)\right]\right) \tag{5.8}$$

2. And the solution to the special case, $x_{k+1} = x_k + a\left[e\left(i_k\right)\right]$ is:

$$x(k) = x_0 + \left(\sum_{j=0}^{k-1} a\left[e\left(i(j)\right)\right]\right) \tag{5.9}$$

3. Similarly the solution to $x_{k+1} = \alpha.x_k + \beta.a\left[e\left(i_{k+1}\right)\right]$ is:

$$x(k) = \alpha^k x_0 + \beta\left(\sum_{j=0}^{k-1} \alpha^j a\left[e\left(i(k-j)\right)\right]\right) \tag{5.10}$$

4. And the solution to the special case, $x_{k+1} = x_k + a\left[e\left(i_{k+1}\right)\right]$ is:

$$x(k) = x_0 + \left(\sum_{j=1}^{k} a\left[e\left(i(j)\right)\right]\right) \tag{5.11}$$

As an example of solving difference equations for sums of sequences, consider the loop:

```
i := 0; sum := 0;
do i <= N ->

     sum := sum + a[i];
     i := i + 1;

od;
```

The assignment `i:=i+1` yields the difference equation $i_{k+1} = i_k + 1$. By (5.2) we establish that this difference equation has the solution $i(k) = k$, so $i(k+1) = k+1$.

The assignment `sum:=sum+a[i]` occurs before the assignment to i in the body of the loop, it therefore yields the difference equation $sum_{k+1} = sum_k + a\left[i_k\right]$. By (5.9) we establish that this difference equation has the solution

$$\begin{aligned} sum(k) &= sum_0 + \left(\sum_{j=0}^{k-1} a[i(j)]\right) \\ &\equiv \sum_{j=0}^{k-1} a[j] \end{aligned}$$

Therefore, $i_{k+1} = i(k+1) = \sum_{j=0}^{k} a[j]$.

Solutions to products of sequences

Similarly product equations of the form $x_{k+1} = \alpha i_k x_k + \beta$ (case1) and $x_{k+1} = \alpha i_{k+1} x_k + \beta$ (case2) can be solved as:

Case 1: Variable Multiplied Prior to Changing

$$\begin{aligned}
x_{k+1} &= \alpha i_k x_k + \beta \\
&= \alpha i_k \left(\alpha i_{k-1} x_{k-1} + \beta\right) + \beta \\
&= \alpha^2 i_k i_{k-1} x_{k-1} + \alpha i_k \beta + \beta \\
&= \alpha^2 i_k i_{k-1} x_{k-1} + \beta \left(\alpha i_k + 1\right) \\
&= \alpha^2 i_k i_{k-1} \left(\alpha i_{k-2} x_{k-2} + \beta\right) + \beta \left(\alpha i_k + 1\right) \\
&= \alpha^3 i_k i_{k-1} i_{k-2} x_{k-2} + \beta \left(\alpha^2 i_k i_{k-1} + \alpha i_k + 1\right) \\
&. \\
&. \\
&. \\
&= \alpha^{k+1} \left(i_k i_{k-1} ... i_0\right) x_0 + \beta \left(\alpha^k \left(i_k i_{k-1} ... i_1\right) + \alpha^{k-1} \left(i_k i_{k-1} ... i_2\right) + ... + \alpha i_k + 1\right) \\
&= \left(\alpha^{k+1} x_0 \prod_{j=0}^{k} i\left(j\right)\right) + \beta \left(\sum_{j=0}^{k} \left(\alpha^j \prod_{m=1}^{j} i\left(k-m+1\right)\right)\right)
\end{aligned}$$

So, the solution to an equation of the form $x_{k+1} = \alpha i_k x_k + \beta$ is

$$x(k) = \left(\alpha^k x_0 \prod_{j=0}^{k-1} i\left(j\right)\right) + \beta \left(\sum_{j=0}^{k-1} \left(\alpha^j \prod_{m=1}^{j} i\left(k-m\right)\right)\right) \tag{5.12}$$

In most cases, α will equal 1 and β will equal 0, so the iterative form will be $x_{k+1} = i_k x_k$, which has the solution

$$\begin{aligned}
x(k) &= \left(1^k x_0 \prod_{j=0}^{k-1} i\left(j\right)\right) + 0. \left(\sum_{j=0}^{k-1} \left(1^j \prod_{m=1}^{j} i\left(k-m\right)\right)\right) \\
&= \left(x_0 \prod_{j=0}^{k-1} i\left(j\right)\right) \qquad (5.13)
\end{aligned}$$

We can now calculate strong invariant properties for loops such as the following using (5.1) and (5.13):

```
i:=1;prod:=1;
do i <= 100 ->
    prod := i*prod;
    i := i + 1;
od;
```

By (5.2), $i(k) = k+1$, so i after a $k+1$st iteration is $i(k+1) = k+1+1 = k+2$, so by (5.13), *prod* after the same iteration can be calculated as:

$$\begin{aligned}
prod_{k+1} &= prod(k+1) \\
&= prod_0 \left(\prod_{j=0}^{k} i(j) \right) \\
&= 1. \left(\prod_{j=0}^{k} i(j) \right) \\
&= \prod_{j=0}^{k} i(j) \\
&= \prod_{j=0}^{k} (j+1) \\
&= \prod_{j=1}^{k+1} j
\end{aligned}$$

Case 2: Variable Multiplied After Changing

We can use equations (5.12) and (5.13) if x is multiplied by i before i is updated in the loop body. If i is updated before being multiplied by x we need solutions for $x_{k+1} = \alpha_{i_k}{}_{+1} x_k + \beta$:

$$\begin{aligned}
x_{k+1} &= \alpha_{i_k}{}_{+1} x_k + \beta \\
&= \alpha^{k+1} \left(i_{k+1} i_k ... i_1 \right) x_0 + \beta \left(\alpha^k \left(i_{k+1} i_k ... i_2 \right) + \alpha^{k-1} \left(i_{k+1} i_k ... i_3 \right) + ... + \alpha i_{k+1} + 1 \right) \\
&= \left(\alpha^{k+1} x_0 \prod_{j=1}^{k+1} i\left(j\right) \right) + \beta \left(\sum_{j=0}^{k} \left(\alpha^j \prod_{m=1}^{j} i\left(k+1-m+1\right) \right) \right) \\
\text{So, } x(k) &= \left(\alpha^{k} x_0 \prod_{j=1}^{k} i\left(j\right) \right) + \beta \left(\sum_{j=0}^{k-1} \left(\alpha^j \prod_{m=1}^{j} i\left(k-m+1\right) \right) \right) \qquad (5.14)
\end{aligned}$$

When α equals 1 and β equals 0, we get:

$$\begin{aligned}
x(k) &= \left(1^{k} x_0 \prod_{j=1}^{k} i\left(j\right) \right) + 0. \left(\sum_{j=0}^{k-1} \left(1^j \prod_{m=1}^{j} i\left(k-m+1\right) \right) \right) \\
&= \left(x_0 \prod_{j=1}^{k} i(j) \right) \qquad (5.15)
\end{aligned}$$

Products of indexed data structures representing sequences

Often the sequences we are multiplying over are indexed data structures such as arrays rather than variables such as i in the above discussion. For example, consider an assignment yielding $x_{k+1} = \alpha.x_k.a\left[e\left(i_k\right)\right] + \beta$ or $x_{k+1} = \alpha.x_k.a\left[e\left(i_{k+1}\right)\right] + \beta$, where i is a variable that is also changing in the body of the loop and e is a function yielding natural numbers (possibly the identity function).

For these special cases the rules for solving these difference equations are:

1. For an assignment yielding the iterative form $x_{k+1} = \alpha.x_k.a\left[e\left(i_k\right)\right] + \beta$ the solution is:

$$x(k) = \left(\alpha^k x_0 \prod_{j=0}^{k-1} a\left[e\left(i(j)\right)\right]\right) + \beta\left(\sum_{j=0}^{k-1}\left(\alpha^j \prod_{m=1}^{j} a\left[e\left(i(k-m)\right)\right]\right)\right) \tag{5.16}$$

2. And the solution to the special case, $x_{k+1} = x_k.a\left[e\left(i_k\right)\right]$ is:

$$x(k) = \left(x_0 \prod_{j=0}^{k-1} a\left[e\left(i(j)\right)\right]\right) \tag{5.17}$$

3. Similarly the solution to $x_{k+1} = \alpha.x_k.a\left[e\left(i_{k+1}\right)\right] + \beta$ is:

$$x(k) = \left(\alpha^k x_0 \prod_{j=1}^{k} a\left[e\left(i(j)\right)\right]\right) + \beta\left(\sum_{j=0}^{k-1}\left(\alpha^j \prod_{m=1}^{j} a\left[e\left(i(k-m+1)\right)\right]\right)\right) \tag{5.18}$$

4. And the solution to the special case, $x_{k+1} = x_k.a\left[e\left(i_{k+1}\right)\right]$:

$$x(k) = \left(x_0 \prod_{j=1}^{k} a\left[e\left(i(j)\right)\right]\right) \tag{5.19}$$

Special Cases

There are two special case difference equations required for completeness. The first occurs when a variable is not updated in the body of the loop. Such difference equations have the iterative form, $x_{k+1} = x_k$. Difference equations of this form have the solution

$$x(k) = x_0 \tag{5.20}$$

The second is the difference equation produced as a result of a redundant assignment of a constant value to a variable within the body of a loop. Such difference equations will be of the form $x_{k+1} = \alpha$, where α is a constant. They have the solution

$$x(k) = \alpha \tag{5.21}$$

Summary

Table 5.1 is a summary of the solutions derived above. A tool for automating the generation of invariants could use such a table to match iterative form difference equations to solutions.

Table 5.1: Difference Equation Solutions

Iterative Form Equation	**Solution**
1. $x_{k+1} = \alpha x_k + \beta$	$x(k) = \begin{cases} x_0 + \beta k, \text{ when } \alpha = 1 \\ \alpha^k x_0 + \beta\left(\frac{1-\alpha^k}{1-\alpha}\right), \text{ otherwise} \end{cases}$
2. $x_{k+1} = {x_k}^n$	$x(k) = {x_0}^{\left(n^k\right)}$
3. $x_{k+1} = \alpha x_k + \beta i_k$	$x(k) = \begin{cases} x_0 + \left(\sum_{j=0}^{k-1} i(j)\right), \text{ when } \alpha \text{ and } \beta\ = 1 \\ \alpha^k x_0 + \beta\left(\sum_{j=0}^{k-1} \alpha^j i(k-j-1)\right), \text{ otherwise} \end{cases}$
4. $x_{k+1} = \alpha x_k + \beta i_{k+1}$	$x(k) = \begin{cases} x_0 + \left(\sum_{j=0}^{k} i(j)\right), \text{ when } \alpha \text{ and } \beta\ = 1 \\ \alpha^k x_0 + \beta\left(\sum_{j=0}^{k-1} \alpha^j i(k-j)\right), \text{ otherwise} \end{cases}$
5. $x_{k+1} = \alpha x_k + \beta . a\left[e\left(i_k\right)\right]$	$x(k) = \begin{cases} x_0 + \left(\sum_{j=0}^{k-1} a[e(i(j))]\right), \text{ when } \alpha \text{ and } \beta\ = 1 \\ \alpha^k x_0 + \beta\left(\sum_{j=0}^{k-1} \alpha^j a[e(i(k-j-1))]\right), \text{ otherwise} \end{cases}$
6. $x_{k+1} = \alpha x_k + \beta . a\left[e\left(i_{k+1}\right)\right]$	$x(k) = \begin{cases} x_0 + \left(\sum_{j=1}^{k} a[e(i(j))]\right), \text{ when } \alpha \text{ and } \beta\ = 1 \\ \alpha^k x_0 + \beta\left(\sum_{j=0}^{k-1} \alpha^j a\left[e(i(k-j))\right]\right), \text{ otherwise} \end{cases}$
7. $x_{k+1} = \alpha . i_k . x_k + \beta$	$x(k) = \begin{cases} x_0\left(\prod_{j=0}^{k-1} i(j)\right), \text{ when } \alpha = 1 \text{ and } \beta\ = 0 \\ \left(\begin{array}{c} \left(\alpha^k x_0 \prod_{j=0}^{k-1} i(j)\right) + \\ \beta\left(\sum_{j=0}^{k-1}\left(\alpha^j \prod_{m=1}^{j} i(k-m)\right)\right) \end{array}\right), \text{ otherwise} \end{cases}$
8. $x_{k+1} = \alpha . i_{k+1} . x_k + \beta$	$x(k) = \begin{cases} x_0\left(\prod_{j=1}^{k} i(j)\right), \text{ when } \alpha = 1 \text{ and } \beta\ = 0 \\ \left(\begin{array}{c} \left(\alpha^k x_0 \prod_{j=1}^{k} i(j)\right) + \\ \beta\left(\sum_{j=0}^{k-1}\left(\alpha^j \prod_{m=1}^{j} i(k-m+1)\right)\right) \end{array}\right), \text{ otherwise} \end{cases}$
9. $x_{k+1} = \alpha . a\left[e\left(i_k\right)\right] . x_k + \beta$	$x(k) = \begin{cases} x_0\left(\prod_{j=0}^{k-1} a\left[e\left(i\left(j\right)\right)\right]\right), \text{ when } \alpha = 1 \text{ and } \beta\ = 0 \\ \left(\begin{array}{c} \left(\alpha^k x_0 \prod_{j=0}^{k-1} a\left[e\left(i\left(j\right)\right)\right]\right) + \\ \beta\left(\sum_{j=0}^{k-1}\left(\alpha^j \prod_{m=1}^{j} a\left[e\left(i\left(k-m\right)\right)\right]\right)\right) \end{array}\right), \text{ otherwise} \end{cases}$
10. $x_{k+1} = \alpha . a\left[e\left(i_{k+1}\right)\right] . x_k + \beta$	$x(k) = \begin{cases} x_0\left(\prod_{j=1}^{k} a\left[e\left(i\left(j\right)\right)\right]\right), \text{ when } \alpha = 1 \text{ and } \beta\ = 0 \\ \left(\begin{array}{c} \left(\alpha^k x_0 \prod_{j=1}^{k} a\left[e\left(i\left(j\right)\right)\right]\right) + \\ \beta\left(\sum_{j=0}^{k-1}\left(\alpha^j \prod_{m=1}^{j} a\left[e\left(i\left(k-m+1\right)\right)\right]\right)\right) \end{array}\right), \text{ otherwise} \end{cases}$
11. $x_{k+1} = x_k$	$x(k) = x_0$
12. $x_{k+1} = \alpha$	$x(k) = \alpha$

Where, x and i are scalar variables and $i_{k+1} \neq i_k$, $a[]$ is an indexed (array) variable, and n, α and β are constants.

5.2 Invariant Calculation

The generation of invariants from non-branching loop bodies, using the difference equation solutions described above, is now discussed. Existing methods of generating invariants [9, 62] are extended by describing an algorithmic approach to generating universally quantified invariant formulae. The method described in this section is only applicable when all iterative form difference equations in $sp\,(Q_{k+1}, S)$ can be solved. In chapter 9, we discuss the use of iterative invariant representations for reasoning about program correctness, when difference equations can not be solved.

In this discussion on calculating invariants we will continue the exponentiation example of Program 2.

5.2.1 Using the Difference Equation Solutions

For any loop of the form $\{Q\}\texttt{do}\ \ G \rightarrow S\ \ \texttt{od}$, if we calculate $R_{k+1} = sp\,(Q_{k+1},\ S)$, R_{k+1} will be of the form

$$R_{k+1} \equiv (k \in \mathbb{N} \wedge v1_{k+1} = f_{v1}\,(v1_k) \wedge \ldots \wedge vn_{k+1} = f_{vn}\,(vn_k))$$

where $v1, ..., vn$ are variables and $f_{v1}, ..., f_{vn}$ are functions. R_{k+1} can be calculated directly because we impose a fixed calculation depth of 1. The terms $v_{k+1} = f_v\,(v_k)$ describe difference equations in iterative form, yielded by loop body assignments.

We make use of the property, that for any variable v whose value on an arbitrary $k + 1$st iteration is described by the iterative form difference equation $v_{k+1} = f\,(v_k)$, then if this difference equation can be solved, the solution will be represented by $v(k)$, and $v_{k+1} = v(k + 1)$.

R_{k+1} is refined by (5.22) for all variables v, with an iterative difference equation term in R_{k+1} of the form $v_{k+1} = f\,(v_k)$, having the difference equation solution $v(k)$.

$$R_{k+1} \sqsubseteq ((R_{k+1}\,[v(k+1)/f\,(v_k)])\,[v(k)/v_k]) \tag{5.22}$$

Lemma 5.1. Refinement (5.22) is equivalent to the following:

$$R_{k+1} \sqsubseteq ((R_{k+1}\,[v(k)/v_k])) \tag{5.23}$$

Proof. For any iterative-form difference equation $v_{k+1} = f\,(v_k)$ having the solution $v(k)$, $v(k+1) = f\,(v(k))$. Therefore, $R_{k+1}\,[v(k)/v_k] \equiv R_{k+1}\,[v(k+1)/f\,(v_k)]\,[v(k)/v_k]$. □

Both forms of substitution have benefit. The former may be easier to automate as it is only reliant on pattern matching and substitution. The later is easier to manually apply, although, algebraic simplification is required to obtain the simplified results that the former provides.

Consider Program 2. In section 4.1.2, we calculated the iterative form invariant R_{k+1} to be

$$\begin{aligned} R_{k+1} &= sp\,(Q_{k+1},\ r := r * x; y := y - 1) \\ &\equiv (k \in \mathbb{N} \wedge r_{k+1} = r_k x_k \wedge x_{k+1} = x_k \wedge y_{k+1} = y_k - 1 \wedge y_k > 0) \end{aligned}$$

We know by examining the loop precondition, Q, that $y_0 = Y,\ x_0 = X$ and $r_0 = 1$.

The iterative form difference equations described by R_{k+1} are:

- $y_{k+1} = y_k - 1$
- $x_{k+1} = x_k$
- $r_{k+1} = r_k x_k$

Their respective solutions, obtained by looking up Table 5.1, are:

- $y(k) = Y - k$
- $x(k) = X$
- $r(k) = \prod_{j=0}^{k-1} x(j) = \prod_{j=0}^{k-1} X = X^k$

Applying (5.22) to R_{k+1} using these difference equation solutions, gives:

$$R_{k+1} \sqsubseteq \Big(k \in \mathbb{N} \wedge r_{k+1} = X^{k+1} \wedge x_{k+1} = X \wedge y_{k+1} = Y - k - 1 \wedge Y - k > 0\Big)$$

All of the difference equations could be solved. R_{k+1} now describes the values of the program variables on an arbitrary $k + 1$st iteration as functions of the variables' initial values and the iterative count. These equalities are invariant. For any value of $k \geq 0$, they will hold.

5.2.2 Toward a More Readable Invariant

As mentioned above, R_{k+1} is invariant for any $k \geq 0$, because R_{k+1} has been calculated for any arbitrary $k+1$st iteration. We can use this property to make the invariant more human readable to support reasoning about correctness

Because the relationships implied by R_{k+1} hold on every iteration, so long as *all* iterative form difference equations have been solved, we can drop the subscripts on the variables in R_{k+1}. This refinement produces a predicate that is independent of iteration specific variable names. We denote the resulting invariant, P.

Dropping the subscripts in the exponentiation example gives us the following invariant P:

$$P \equiv \Big(k \in \mathbb{N} \wedge r = X^{k+1} \wedge x = X \wedge y = Y - k - 1 \wedge Y - k > 0\Big)$$

Removing References to the Iteration Count from the Invariant

To improve the readability and understandability of P, an attempt is made to remove all references to the iteration count k in the invariant by making a substitution. In general, if there is an equality of the form $v = f(k)$ in P, for some variable v, and we can find f^{-1}, such that $k = f^{-1}(v)$, then we can remove k from P by applying the substitution

$$P\left[f^{-1}(v)/k\right] \tag{5.24}$$

Applying (5.24) to the exponentiation example, using the fact $(y = Y - k - 1) \equiv (k = Y - y - 1)$, gives the modified invariant below:

$$P \quad \equiv \quad \left(Y - y - 1 \in \mathbb{N} \wedge r = X^{Y-y} \wedge x = X \wedge y \geq 0\right)$$

This removes the introduced variable k from our specification, and leaves us with a strong specification of the invariant properties of the exponentiation loop after iterating at least once.

Handling Uninitialised Variables

If P contains references to v_0 for any variable v, if there is an equality in the loop precondition, Q, of the form $v = e$ or $e = v$, where e is an expression, then substitution (5.25) can be applied to remove all references to v_0.

$$P = P\left[e/v_0\right], \text{ for all variables } v \in P \text{ where } Q \Rightarrow (v = e) \tag{5.25}$$

For a variable v, if P contains a reference to v_0 and there is no expression e such that $Q \Rightarrow (v = e)$, then after applying (5.25), references to v_0 will remain in P. Such references indicate that v is uninitialised and is used before being initialised. Whenever v_0 remains in P then there is a solution to a difference equation that has required a non-determinable initial value of v. Because P is dependent on v_0, then the uninitialised variable v is required to describe the semantics of the loop being analysed in a deterministic manner.

The failure to initialise a variable prior to it being required is a defect which must be noted. Any automated system realising this approach to deriving invariants must automatically generate such notes as warnings to a verifier. Such automation could involve a search for remaining v_0s in P.

Such an interpretation of uninitialised variables is consistent with the memory models employed by all programming languages that do not trap the use of uninitialised variables in the compilation/interpretation process. That is, an uninitialised variable, v, will have some value stored in its assigned memory location. If v has not been initialised by the program, then that value depends on the compiler that the program being analysed is compiled with and the initial state of the memory that the program will be run under. For example, some compilers/interpreters may automatically initialise all uninitialised integers to zero. Others may not do this, meaning that the initial value of an integer variable is the integer representation of whatever was in its memory location to begin with. The notation v_0 represents the value of whatever is stored in the memory location for variable v prior to iteration of the loop being analysed. A specification that involves v_0 references that cannot be removed describes a non-deterministic algorithm.

In summary, we define the following inspection rule to identify uninitialised variables:

Inspection Rule 1 (Investigate Uninitialised Variables). For any loop, if we calculate P and there are remaining references in P to initial values, v_0 for any variable v, then v is uninitialised.

This gives rise to the following inspection question:

> Are there any references to v_0 for any variable v in the variable set of the loop body and loop guard?

Not Solvable Difference Equations In some cases, solutions to the difference equations in R_{k+1} can not be found or are difficult to calculate. In such cases (5.22) can not be applied. In many of these cases, however, presenting R_{k+1} in terms of unsolved difference equations is useful for human verification. This was seen with the algorithm for implementing the Newton method of approximating square root in the previous chapter. It may be beneficial for reading purposes to present the solvable equations in solved form and the remainder in iterative form.

5.3 Calculating the Postcondition

The strongest postcondition for iterative constructs is defined in [3, 24] as

$$sp(Q, do\, G \rightarrow S\; od) \;=\; (\neg G \wedge (Q \vee P_1 \vee ... \vee P_m))$$

where m is a positive integer representing the number of iterations to termination, and $P_1 = sp(Q \wedge G, S)$ and $P_{i+1} = sp(P_i \wedge G, S)$ for $0 < i \leq m$. The invariant P, calculated by the method above, is equivalent to $(P_1 \vee ... \vee P_m)$. Using this equivalence, the definition above can be modified to the following closed form, calculable whenever P can be obtained:

$$sp(Q, do\, G \rightarrow S\; od) \;=\; (\neg G \wedge (Q \vee P)) \tag{5.26}$$

Applying (5.26) to the exponentiation example, the post condition is:

$$
\begin{aligned}
R &= (\neg G \wedge (Q \vee P)) \\
&\equiv \left(y \leq 0 \wedge \begin{pmatrix} X \in \mathbb{N} \wedge Y \in \mathbb{N} \wedge y = Y \wedge r = 1 \wedge x = X \vee \\ Y - y - 1 \in \mathbb{N} \wedge y > -1 \wedge r = X^{Y-y} \wedge x = X \end{pmatrix}\right) \\
&\equiv \begin{pmatrix} X \in \mathbb{N} \wedge Y \in \mathbb{N} \wedge y = 0 \wedge y = Y \wedge r = 1 \wedge x = X \vee \\ y = 0 \wedge Y - y - 1 \in \mathbb{N} \wedge r = X^{Y-y} \wedge x = X \end{pmatrix} \\
&\equiv \left(x = X \wedge r = X^{Y-y} \wedge y = 0 \wedge X \in \mathbb{N} \wedge \begin{pmatrix} y = Y & \vee \\ Y - 1 \in \mathbb{N} & \end{pmatrix}\right) \\
&\equiv \left(x = X \wedge r = X^{Y} \wedge y = 0 \wedge X \in \mathbb{N} \wedge Y \in \mathbb{N}\right) \qquad (5.27)
\end{aligned}
$$

This postcondition is strong enough to be used in a formal proof of correctness, or for reasoning about the exponentiation program. The specification for this program states that $r = X^Y$ shall hold on termination. The derived postcondition, R implies the verification condition $r = X^Y$. Program 2 establishes $r = X^Y$ under the given precondition, $(X, Y \in \mathbb{N} \wedge y = Y \wedge x = X)$. It is therefore partially correct.

We define the following inspection rule:

Inspection Rule 2 (Partial Correctness). Let R be the postcondition calculated for a program `S`, with a given precondition Q_{given} and a given postcondition R_{given}, where $R \equiv sp\,(Q_{given}, \mathtt{S})$. If $R \implies R_{given}$, then S is partially correct with respect to Q_{given} and R_{given}.

This gives rise to the following inspection question:

> Is the program correct with respect to a given specification? That is, does the calculated postcondition for the program under the given precondition imply the given postcondition?

The prototype tool supports the answering of this inspection question by stating the direct proof obligations for demonstrating partial correctness (see Fig. **??** for a good example).

The above inspection question is answerable by human reasoning even when the given specification is not formally stated. As an example of this, consider the Naur Text Reformatting example of the previous chapter. With the derived information, we were able to reason about partial correctness with respect to a non-formal specification. This presents a distinct advantage over current formal methods of verification, which are restricted to verification entirely within a formal system of specification.

The remaining sections of this chapter present specialised rules for generating quantified invariants yielded from programs involving sequence data types[4].

[4] In order to simplify the discussion in this thesis, we consider that all sequential data structures can be considered as dynamic arrays. Hence, we present all sequential data structures using array notation, where indexing begins at 0.

5.4 Quantification and Sequence Properties

Of the statements that must be analysed in the bodies of loops, assignments and guards involving non-scalar sequential data structures require special consideration. Assignments and guards involving non-scalar variables yield quantified semantics. Most existing techniques for deriving invariants and postconditions fail to completely define these semantics. This section discusses the many issues associated with deriving semantics from statements involving sequences and presents an algorithm for performing the derivation. This is a complex algorithm. Heuristic identification of common simple classes of problems involving sequences are discussed and simplified rules are presented for the calculation of semantics for these simpler problems. For the remainder of this section sequences will be referred to as arrays.

5.4.1 Heuristic for Determining the Type of Array Assignment

Consider the loop $\{Q\}$ `do` $G \;\rightarrow\; S$ `od`. Array variables are assigned values in S in one of two ways. An assignment can be made to a fixed element of an array in every iteration. In this case the fixed array element can be treated as any scalar variable, and Algorithm 1 for calculating strongest postcondition can be applied.

The more complex case occurs when an assignment in S is made to an array element that is indexed dependent on variables that are changing in S. As an example, consider the trivial array initialisation loop, `i:=0; do i < N` $\rightarrow$ `a[i]:=0; i:=i+1; od`. The semantics of such assignments must be described by quantified formulae.

An assignment to an array variable will be of the form $a[e] := exp$, where e and exp are expressions. If R_{k+1} is calculated, as usual, by $R_{k+1} = sp\,(Q_{k+1},\ S)$, then the assignment $a[e] := exp$ in S will correspond to a predicate $a[e'] = exp'$ in R_{k+1} after refinement (5.22).

Definition 5. If $k \notin \nu\,(e')$, then $a[e]$ refers to a fixed element in every iteration. Such an assignment will be called a *fixed element array assignment*, and $a[e]$ can be treated as any other scalar variable.

Definition 6. If $k \in \nu\,(e')$, then the index of the array element being assigned is dependent on a function, f, of the iteration count k, where $f(k) = e'$. Such an assignment will be called an *iteration-dependent array assignment* and $a[e]$ can not be treated as a scalar variable.

Consider the example of an assignment to a fixed array element in Program 3. The initial conditions on execution of the loop body on an arbitrary $k+1$st iteration, are given by:

$$Q_{k+1} = (k \in \mathbb{N} \wedge a[0]_{k+1} = a[0]_k \wedge i_{k+1} = i_k \wedge i_k < N)$$

Program 3 Example of fixed array element assignment in the body of a loop

```
i:=0; a[0]:=0;
do i < N →
    a[0]:=a[0]+i;
    i:=i+1;
od;
```

Applying the strongest postcondition transformer to Q_{k+1} and the loop body gives:

$$\begin{aligned} R_{k+1} &= sp\,(Q_{k+1},\ a[0] := a[0] + i;\ i := i + 1) \\ &\equiv (k \in \mathbb{N} \wedge a[0]_{k+1} = a[0]_k + i_k \wedge i_{k+1} = i_k + 1 \wedge i_k < N) \end{aligned}$$

The indexing expression in the equality involving a is independent of the iteration count, so treating $a[0]$ as a scalar variable and applying refinement (5.22), R_{k+1} becomes:

$$R_{k+1} = \left(k \in \mathbb{N} \wedge a[0]_{k+1} = \sum_{j=0}^{k} j \wedge i_{k+1} = k + 1 \wedge k < N \right)$$

After dropping the subscripts, the invariant after at least one iteration is given by:

$$P = \left(i \geq 1 \wedge a[0] = \sum_{j=0}^{i-1} j \wedge i - 1 < N \right)$$

5.4.2 Iteration-dependent Array Assignments

Consider the array initialisation loop, `i:=0; do i<N → a[i]:=0; i:=i+1; od`. Intuitively most inspectors could determine that this loop initialises all of the elements of an array of length N to be zero, but the method presented in this thesis requires extension to be able to derive this quantified information.

Applying the procedure of the previous section, R_{k+1} is calculated and refined by (5.22) to give

$$R_{k+1} = \left(k \in \mathbb{N} \wedge i_{k+1} = k + 1 \wedge a\,[k]_{k+1} = 0 \wedge k < N \right)$$

We see that the index of the array a in the equality $a[k]_{k+1} = 0$ is dependent on the iteration count, k. That is, by the definitions given above, the assignment yielding this predicate is an iteration-dependent array assignment. Because this loop contains no branching structures, it can be reasoned that the previous k iterations would have established $a[k-1] = 0 \wedge a[k-2] = 0 \wedge ... \wedge a[0] = 0$, respectively. Generalising, it can be asserted that at the end of the $k + 1$st iteration, $\forall j \in [0..k](a[j] = 0)$.

Quantification over the iteration count is very powerful and always applicable. In effect, by induction we can determine that if some predicate of the form $a[e(k)]$ `op` exp is true after an arbitrary $k+1$st iteration, where $e(k)$ is an expression involving k and `op` is a binary operator, then it will have held after all previous iterations. This is true because the loop is non-branching and, therefore, must always iterate through the same path. Formalising, we can inductively state that for a relation of the form $a[e(k)]$ `op` exp, where $k \geq 0$, the following is invariant:

$$\forall j \in [0..k]\,(a[e(j)]\ \texttt{op}\ \ exp[j/k]) \tag{5.28}$$

This is adequate for deriving array semantics in simple cases. However, this is not always sufficient, even for dealing with array assignments in a non-branching loop bodies. For example, if the indexing expression used in an array assignment defines a parabolic function, then this function may map program variables onto the same natural number index in more than one iteration. In such a case, (5.28) will result in a contradiction. For example, in the section 5.4.3, we consider the assignment $\texttt{a}\left[(\mathrm{i}-10)^2\right]\texttt{:=i}$ in the body of a loop that increments `i` by 1 with every iteration. This yields the equality $a\left[(\mathrm{i}-10)^2\right] = i$ in the iterative-form invariant. Applying (5.28) to this iterative-form results in a quantified predicate of the form $\forall j \in [0..k]\left(a\left[(j-10)^2\right] = j\right)$. Depending on the value of the iteration count k, this predicate describes contradictions. For example, assume that $k > 10$, then the above predicate states that when $j = 9$, $a[1] = 9$, and also that $a[1] = 11$ when $j = 11$.

There are also additional problems with (5.28) in that it fails to describe the semantics of the following situation. Consider two iteration-dependent array assignments in the body of a loop, $\texttt{a[e]:=exp}_1$ and $\texttt{a[f]:=exp}_2$. It is possible that the index generated from the index expression e on iteration k is equal to the index generated from the index expression f on a subsequent iteration, $k+C$. Hence, one assignment may overwrite the value assigned by the other assignment, making the overwritten assignment redundant for certain iterations. This situation is usually indicative of a poorly written loop, however, the method of derivation should be robust enough to deal with it. As such we must extend (5.28). An example will illustrate the problem.

Consider the loop, `i := 1; do i < N` $\rightarrow$ `a[i+1]:=A; a[i-1]:=B; i:=i+1; od;`. The array elements assigned with the `i+1` index expression on an iteration j are re-assigned by the assignment with the `i-1` index expression on iteration $j+2$ for all but the last two iterations, k and $k+1$.

The next section deals with extending (5.28) to provide for the calculation of quantified formulae to describe array assignment semantics for all non-branching loop bodies, including those involving rewrites to array elements.

5.4.3 Deriving Array Assignment Semantics

Consider a loop of the form:

```
S_init ;
do G →
    S_0 ;
    a[ex_1 (v_1)] := exp_1 ;
    ...
    a[ex_n (v_n)] := exp_n ;
    S_n ;
od
```

where $ex_1(v_1)..ex_n(v_n)$ are functions of the variable vectors $v_1..v_n$, respectively, $exp_1..exp_n$ are expressions not involving iterative dependent references to elements of a, and S_{init}, $S_0..S_n$ are statements (possibly `skip`), and $S_0..S_n$ do not involve assignment to an element of the array a. Note the sequential numbering of array index functions from 1 to n as they occur in the code.

Let R_{k+1} after refinement (5.22) be a formula of the form:

$$R_{k+1} \equiv (a[e_1(k)] = exp_1 \wedge ... \wedge a[e_n(k)] = exp_n \wedge F),$$

where $e_1(k), ..., e_n(k)$ are functions from the set of natural numbers to the set of natural numbers and are derived from the array index expressions $ex_1(v_1)..ex_n(v_n)$, respectively, and F is a formula.

For every $e_i(k)$, where $i \in [1..n]$, if there is an $e_j(k)$, where $j \in [1..n]$, such that $e_i(k+C) = e_j(k)$, where $C \in \mathbb{N}^+$, then the assignment $\mathtt{a}[e_j(v_j)]\mathtt{:=}exp_j$ is overwritten and made redundant by the assignment $\mathtt{a}[e_i(v_i)]\mathtt{:=}exp_i$ every C iterations.

Redundant Iterations

Redundancy in assignment to sequence variables must be taken into account when describing the semantic properties yielded from loops assigning to sequences. In order to establish the overwrite pattern of iteration-dependent array assignments in a loop, it is necessary to define the set of iterations in which a value is written to an array element at a particular index which gets overwritten on future iterations. We can then use this set of iterations to identify which indexed array elements are redundantly assigned.

Let $O_{p,q}$ be the subset of values for k, the iteration count, where an assignment $a[ex_q(v_q)] := exp_q$ on the $k+1$st iteration is made redundant by the assignment $a[ex_p(v_p)] := exp_p$ on some $(k+1+C)$th iteration, where $0 \leq k \leq k+C$.

$O_{p,q}$ is defined formally by

$$O_{p,q} = \{j \in [0..k] \,|\, \exists l \in [j..k]\,(e_p(l) = e_q(j))\} \tag{5.29}$$

The special case where an assignment makes itself redundant in future iterations must also be considered. $O_{p,p}$ for all $p \in [1..n]$ is defined as:

$$O_{p,p} = \{j \in [0..k] \,|\, \exists l \in [j+1..k]\,(e_p(l) = e_p(j))\} \tag{5.30}$$

For example, consider the loop `i:=0; do i<20` $\rightarrow$ `a`$\left[(\mathrm{i}-10)^2\right]$`:=i; i:=i+1; od`. The assignment `a`$\left[(\mathrm{i}-10)^2\right]$`:=i` on iterations 2 through 10 is made redundant by the same assignment on iterations 12 through 20 respectively.

$$\begin{aligned}
O_{1,1} &= \left\{ j \in [0..k] \,\middle|\, \exists l \in [j+1..k] \left((l-10)^2 = (j-10)^2\right) \right\} \\
&\equiv \{ j \in [0..k] \,|\, \exists l \in [j+1..k] \left(|(l-10)| = |(j-10)|\right) \} \\
&\equiv \{ j \in [0..k] \,|\, \exists l \in [j+1..k] \left((l-10) = (10-j)\right) \} \\
&\equiv \{ j \in [0..k] \,|\, \exists l \in [j+1..k] \left(l = 20-j\right) \}
\end{aligned}$$

As an example of calculating the overwrite sets for a sequence of array assignments in a loop body, consider the loop in Program 4.

Program 4 A loop with redundant array assignments

```
i := 0;
do i < N →

     a[2i]:=A;
     a[i+2]:=B;
     i:=i+2;

od
```

Following the method of the previous section, R_{k+1} can be calculated and refined by (5.22) to yield

$$P = (k \in \mathbb{N} \wedge a[4k] = A \wedge a[2k+2] = B \wedge i = 2k+2 \wedge 2k < N)$$

Numbering the iteration-dependent array assignments in sequence, starting from 1, the assignment `a[2i]:=A` is numbered 1, and the assignment `a[i+2]:=B` is numbered 2. So, using the notation above, $e_1(k) = 4k$ and $e_2(k) = 2k+2$.

The set of values of k for which assignment 1 is overwritten by assignment 2 is given by $O_{2,1}$, where:

$$\begin{aligned}
O_{2,1} &= \{ j \in [0..k] \,|\, \exists l \in [j..k] \left(e_2(l) = e_1(j)\right) \} \\
&\equiv \{ j \in [0..k] \,|\, \exists l \in [j..k] \left(2l+2 = 4j\right) \} \\
&\equiv \{ j \in [0..k] \,|\, \exists l \in [j..k] \left(l = 2j-1\right) \} \\
&\equiv \{ j \in [0..k] \,|\, j \leq 2j-1 \leq k \} \\
&\equiv \left\{ j \in [0..k] \,\middle|\, 1 \leq j \leq \frac{k+1}{2} \right\} \\
&\equiv \left[1..\frac{k+1}{2} \right]
\end{aligned}$$

The set of values of k for which assignment 2 is overwritten by assignment 1 is given by $O_{1,2}$, where:

$$
\begin{aligned}
O_{1,2} &= \{j \in [0..k] \,|\exists l \in [j..k]\,(e_1(l) = e_2(j))\} \\
&\equiv \{j \in [0..k] \,|\exists l \in [j..k]\,(4l = 2j + 2)\} \\
&\equiv \left\{j \in [0..k] \,\middle|\exists l \in [j..k]\left(l = \frac{j+1}{2}\right)\right\} \\
&\equiv \left\{j \in [0..k] \,\middle| j \leq \frac{j+1}{2} \leq k\right\} \\
&\equiv \left\{j \in [0..k] \,\middle| j \leq 1 \leq \frac{k+1}{2}\right\} \\
&\equiv \{0, 1\}
\end{aligned}
$$

For completeness and investigation of redundancy, we calculate $O_{p,p}$ for all $p \in [1..n]$. In this example, $O_{1,1} = \{\}$ and $O_{2,2} = \{\}$.

Points of Intersection

Notice in the above example that 1 is an element of both $O_{1,2}$ and $O_{2,1}$. This indicates that $(1, e_1(1))$ is a point of intersection of the functions $e_1(j) = 4j$ and $e_2(j) = 2j + 2$. That is, $j = 1$ is a solution to $e_1(j) = e_2(j)$.

Points of intersection are important for modeling the semantics of array assignments in loop bodies. If j is a point of intersection of any two array indexing functions $e_p(k)$ and $e_q(k)$, then the array assignments numbered p and q will both be assigning a value to the same array position in the same iteration, specifically iteration $j + 1$. If the array assignment p occurs sequentially after the array assignment q, that is $p > q$, then $e_p(j)$ overwrites $e_q(j)$ and $a\,[e_p(j)] = exp_p$ holds. Conversely, if $q > p$, $a\,[e_q(j)] = exp_q$ holds. Extending this, if there are $m \geq 2$, e_i's, called $e_{i_1}..e_{i_m}$ where $e_{i_1}(j) = e_{i_2}(j) = ... = e_{i_m}(j)$, then if $p = max\,(i_1...i_m)$, $a\,[e_p(j)] = exp_p$ holds on iteration j.

If non-deterministic language constructs, such as multiple assignments are used, the order of array assignments can not be determined. For a multiple assignment of m assignments to the same array position on the same iteration, any one of the m resulting equalities may hold, meaning that any derived postcondition from the multiple assignment, in such a case, will yield a disjunction of m disjuncts.

Generating the Points of Intersection overwrite set

For deterministic programs, we can describe the semantics of array assignments that write to a single array element in a single iteration by determining the iterations on which an intersection occurs, the array index where the intersection takes place, and the right hand side expression for the overriding assignment on that iteration. This information forms an ordered triple of iterations, point of intersection indices, and overriding values assigned. These ordered triples will be called *Point of Intersection* triples.

In order to describe point of intersection triples, we must first identify all of the iterations where overwrites occur in that single iteration. The first element in the triple provides the value of k for which there is a point of intersection on iteration $k+1$. For each of these ks, we then need to identify which assignment statements in the body of the loop write to the index at the point of intersection in that iteration. The last of these assignments in sequence determines the value assigned to that index.

Each triple for an array a, therefore, is of the form (k, i, v), where $a[i]$ is overwritten with the value v on iteration $k+1$.

The set of point of intersection triples for an array and a particular loop is generated by algorithm 4.

Algorithm 4 Generating the set of points of intersection and their overrides

For a loop yielding the m indexed array equalities, $a\left[e_1(k)\right] = exp_1, ..., a\left[e_m(k)\right] = exp_m$:

1. Generate the set of values of k for which more than one overwrite occurs. This set will be the intersection of all sets $O_{i,j}$, where $i \neq j$. Let S represent this set, then

$$S = \{l\,|l \in O_{i,j} \wedge l \in O_{p,q} \wedge i \neq j \wedge p \neq q \wedge (i \neq p \vee j \neq q)\}$$

2. Next we need to generate for each $l \in S$, the set of indexing expression numbers, p, for which $l \in O_{p,q}$. For all l in S, let S_l be the set of index expression numbers, $p \in [1..m]$, such that:

$$S_l = \{p \in [1..m]\,|\exists q \in [1..m] \exists l \in S\,(p \neq q \wedge l \in O_{p,q})\}$$

3. The set of points of intersection is then given by:

$$POI = \{(l, e_p(l), exp_p)\,|l \in S \wedge p = max\,(S_l)\}$$

Applying algorithm 4 to Program 4: We know that $O_{1,2} = \{0,1\}$ and $O_{2,1} = \left[1..\frac{k+1}{2}\right]$, then:

1. $S = \{0,1\} \cap \left[1..\frac{k+1}{2}\right] = \{1\}$
2. $S_1 = \{1,2\}$
3. $max\,(S_1) = 2$, so $POI = \{(1, e_2(1), exp_2)\} = \{(1,4,B)\}$

This set contains a single point of intersection triple, $(1,4,B)$. Following from the above discussion, this indicates that $a[4] = B$ will hold after iteration 2.

A Formula to Define Array Semantics

Extending (5.28), we propose the following predicate to describe array semantics in the general case.

An iteration-dependent array assignment will yield an equality of the form $a\,[e_p(k)] = exp_p$. By induction, it can be stated that for every iteration $l \in [0..k]$ in which $a\,[e_p(k)]$ is not overwritten in a future iteration, $a\,[e_p(l)] = exp_p$ will hold. The union of all of the overwrite sets for array assignment p, denoted $\bigcup_{q=1}^{n} O_{q,p}$, defines all of the iterations on which $a\,[e_p(k)]$ is overwritten. The array assignment properties yielded from array assignment p can, therefore, be described by

$$\forall l \in \left([0..k] - \bigcup_{q=1}^{n} O_{q,p} \right) (a\,[e_p(l)] = exp_p)$$

The above section also states that points of intersection must also be considered. If for every point of intersection triple for an array, the iteration on which that intersection occurs is prior to the current iteration count, $k+1$, then the point of intersection overwrite must be described in the predicate.

We can use these requirements to specify a formula for describing all of the invariant properties yielded from array assignments. These invariant properties will be denoted P_{as}, where

$$P_{as} = \left(\begin{array}{l} \bigwedge_{p=1}^{n} \left(\forall l \in \left([0..k] - \bigcup_{q=1}^{n} O_{q,p} \right) (a\,[e_p(l)] = exp_p) \right) \wedge \\ \forall (it, x, exp) \in POI\,(it \leq k \Rightarrow a\,[x] = exp) \end{array} \right) \tag{5.31}$$

The initial simple formula (5.28) represents the special case where $O_{p,q} = \emptyset$, for all iteration dependent array assignments p and q to a, and where $POI = \emptyset$ for the array a.

Consider again, Program 4. The overwrite sets are calculated as $O_{1,2} = \{j \in [0..1]\}$ and $O_{2,1} = \left\{ j \in \left[1..\frac{k+1}{2}\right] \right\}$ and the point of intersection overwrite set $POI = \{(1, 4, B)\}$. By (5.31) the array semantic invariant is calculated to be:

$$\begin{array}{rcl} P_{as} & \equiv & \left(\begin{array}{l} \bigwedge_{p=1}^{n} \left(\forall l \in \left([0..k] - \bigcup_{q=1}^{n} O_{q,p} \right) (a\,[e_p(l)] = exp_p) \right) \wedge \\ \forall (it, x, exp) \in POI\,(it \leq k \Rightarrow a\,[x] = exp) \end{array} \right) \\ & \equiv & \left(\begin{array}{l} \forall l \in ([0..k] - (O_{1,1} \cup O_{2,1}))\,(a\,[e_1(l)] = exp_1) \wedge \\ \forall l \in ([0..k] - (O_{1,2} \cup O_{2,2}))\,(a\,[e_2(l)] = exp_2) \wedge \\ k \geq 1 \Rightarrow a[4] = B \end{array} \right) \\ & \equiv & \left(\begin{array}{l} \forall l \in \left([0..k] - \left[1..\frac{k+1}{2}\right]\right) (a\,[4l] = A) \wedge \\ \forall l \in ([0..k] - \{0,1\})\,(a\,[2l+2] = B) \wedge \\ k \geq 1 \Rightarrow a[4] = B \end{array} \right) \\ & \equiv & \left(\begin{array}{l} a[0] = A \wedge \forall l \in \left[\frac{k+1}{2} + 1..k\right] (a\,[4l] = A) \wedge \\ \forall l \in [2..k]\,(a\,[2l+2] = B) \wedge \\ k \geq 1 \Rightarrow a[4] = B \end{array} \right) \\ & \equiv & \left(a[0] = A \wedge \forall l \in \left[\dfrac{k+1}{2} + 1..k \right] (a\,[4l] = A) \wedge \forall l \in [1..k]\,(a\,[2l+2] = B) \right) \end{array}$$

Incorporating P_{as} with Other Invariant Properties

The array assignment equalities, $a\left[e_p(k)\right] = exp_p$, for all $p \in [1..n]$ are dropped from R_{k+1} as they are now described by the invariant P_{as}. R_{k+1} now involves only predicates that do not describe array properties. P_{as} is conjoined to R_{k+1} to give a strong invariant, which we again denote P.

In the example, the equalities $a[4k] = A$ and $a[2k+1] = B$ are described by P_{as}. These equalities are replaced by P_{as}, giving

$$P \equiv \left(\begin{array}{l} k \in \mathbb{N} \wedge i = 2k + 2 \wedge 2k < N \wedge \\ a[0] = A \wedge \forall l \in \left[\frac{k+1}{2} + 1..k\right] (a\,[4l] = A) \wedge \forall l \in [1..k]\,(a\,[2l+2] = B) \end{array} \right)$$

This can then be simplified, by substitution for k, to give

$$P \equiv \left(\begin{array}{l} \frac{i}{2} - 1 \in \mathbb{N} \wedge i - 2 < N \wedge \\ a[0] = A \wedge \forall l \in \left[\frac{i}{4} + 1..\frac{i}{2} - 1\right] (a\,[4l] = A) \wedge \forall l \in [1..\frac{i}{2} - 1]\,(a\,[2l+2] = B) \end{array} \right)$$

This process is repeated for all arrays assigned in the body of the loop.

5.4.4 Persistent Initialisation of Array Variables

A complete specification of an algorithm which manipulates arrays must not only describe changes to the array effected by the algorithm, but also describe the elements of the array that have not changed. In order to begin a discussion on this topic the following convention and notation is introduced.

Convention. The assignment statement `a[] := A[]` means that the entire contents of an array A are assigned to an array a starting from the first array position and ending on the array position which is one less than the greater of $\#a$ and $\#A$. Such assignments will be referred to as *whole array assignments.* The above semantic description of whole array assignment ensures that an assignment of an array to another array does not result in "array out of bounds" type errors. This definition is consistent with our description of arrays as dynamically sized sequences. Algorithm 5 extends the strongest postcondition calculation for whole array assignments.

If a whole array assignment, `a[]:=A[]`, occurs as part of a loop initialisation then the loop precondition Q will imply that $\forall j \in [0..min(\#a, \#A) - 1]\,(a[j] = A[j])$. If such a term is not a conjunct of Q, then it can only be stated that $\forall j \in [0..min(\#a, \#A) - 1]\,(a[j] = a[j]_0)$, where $a[j]_0$ is some unknown initial value of $a[j]$. This assumption becomes important if an uninitialised, indexed array variable occurs in the RHS of an assignment.

In order to describe array properties that are not changed by the body of a loop, the set of all changed array positions must be calculated. This set will be called the *Changed Index Set* of an array.

Algorithm 5 *sp* for whole array assignments

Consider the assignment `a[]:=A[]`, under some precondition Q where a and A are array variables and Q is a formula of the form $Q_1 \vee ... \vee Q_n$ where the sub formulae $Q_1..Q_n$ contain no disjuncts.
$sp(Q, a[] := A[]) = sp(Q_1, a[] := A[]) \vee .. \vee sp(Q_n, a[] := A[])$, where $sp\,(Q_i, a[] := A[])$ is defined for all $i \in [1..n]$ as follows.
For all equalities of the form $a[e] = E$ in Q_i, where a is an array variable and e and E are expressions:

1. If an xC-equality occurs in Q_i, with $a[e]$ being on one side of the equality, then let the constant symbol occurring in the leftmost of the xC-equalities for $a[e]$ be D, then:

   $$sp(Q_i, a[] := A[]) = Q_i[D/a[e]] \wedge \forall j \in [0..min(\#a, \#A) - 1]\,(a[j] = A[j])$$

2. If an xC-equality does not occur in Q_i, and an xv-equality occurs in Q_i, with $a[e]$ being on one side of the equality, let the non-$a[e]$ variable occurring in the leftmost xv-equality of Q_i be w, then:

   $$sp(Q_i, a[] := A[]) = Q_i[w/a[e]] \wedge \forall j \in [0..min(\#a, \#A) - 1]\,(a[j] = A[j])$$

3. If an xC-equality does not occur in Q_i, and an xv-equality does not occur in Q_i, and an xf-equality occurs in Q_i, with $a[e]$ being on one side of the equality, let the non-$a[e]$ term occurring in the leftmost xf-equality of Q_i be u, then:

   $$sp(Q_i, a[] := A[]) = Q_i[u/a[e]] \wedge \forall j \in [0..min(\#a, \#A) - 1]\,(a[j] = A[j])$$

4. Otherwise,

   $$sp(Q_i, a[] := A[]) = Q_i\,[a[e]_0/a[e]] \wedge \forall j \in [0..min(\#a, \#A) - 1]\,(a[j] = A[j])$$

Calculating the Changed Index Set

For a loop yielding R_{k+1}, where $R_{k+1} \Rightarrow (a\,[e_1(k)] = exp_1 \wedge ... \wedge a\,[e_n(k)] = exp_n)$, where $e_1..e_n$ are functions from natural numbers to natural numbers, and $exp_1..exp_n$ are expressions, we can calculate the set of indices of the array a to which the statements of the loop body wrote a value. The index set of a changed by each array index assignment, denoted IC_i, for all $i \in [1..n]$, is given by the set of values of the index expression, e_i, applied to the values of k for which e_i is not overwritten.

$$IC_i = \left\{ p \in \left([0..k] - \bigcup_{j=1}^{n} O_{j,i} \right) \bullet e_i(p) \right\} \tag{5.32}$$

The set of indices of all array elements of a modified by all assignments of a loop body, denoted IC, is given by:

$$IC = \bigcup_{i=1}^{n} IC_i \tag{5.33}$$

The set of all array positions of a not changed by a loop body, called the *Unchanged Index Set* and denoted IU, is therefore given by:

$$IU = [0..\#a - 1] - IC \tag{5.34}$$

Once the unchanged index set has been calculated it is possible to specify the initial array properties not changed by a loop. For any loop and any array, a, with an unchanged index set IU, an invariant called the *Unchanged Array Invariant* and denoted P_{UA}, holds. P_{UA} is defined by (5.35).

$$P_{UA} = \forall i \in IU\,(a[i] = a[i]_0) \tag{5.35}$$

As with P, (5.25) is applied to P_{UA} to remove references to $a[i]_0$ for all $i \in IU$. This substitution is consistent with the discussion of Section 5.2.2.

5.4.5 Array Variables in Assignment RHS

Until now, only assignments to array variables with no iteration-dependent array variables in the expression on the right hand side have been discussed. Consider now assignments of the form `a[e(i)]:=exp`, where i is an index variable ($i(k) \neq i_0$), $e(i)$ is a function of i yielding a natural number, and *exp* is an expression involving at least one term of the form `a[f(j)]`, where j is an index variable and $f(j)$ is a function of j yielding a natural number. Such assignments pose additional problems. As an example of such an assignment, consider `a[i+1]:=2*a[i]` in the following loop:

$\{\forall j \in [0..N]\,(a[j] = A[j]) \wedge i = 0\}$ The constant equality $N = \#a - 1$ also holds
`do i<N` $\rightarrow$

```
        a[i+1]:=2*a[i];
        i:=i+1;
    od;
```

After the first iteration, $a[1] = 2 * a[0]$, after the second iteration, $a[2] = 2 * a[1] = 2 * 2 * a[0] = 2^2 a[0]$, continuing after the kth iteration, $a[k] = 2^k a[0]$. Applying the method described in this chapter there will be a term in R_{k+1} of the form $a[k+1]_{k+1} = 2a[k]_k$, which will yield $\forall j \in [0..k]\,(a[j+1] = 2a[j])$. This recursively defines the progression $\forall j \in [0..k]\,\big(a[j+1] = 2^{j+1}a[0]\big)$. Defining all progressions in this non-recursive form is generally useful for verification.

Algorithm 6 defines heuristic rules to handle array predicates describing progressions.

Algorithm 6 Heuristic rules for array predicates describing progressions

1. For predicates in the iterative-form invariant of the form:

$$a\left[e_i(k)\right]_{k+1} = g\left(a\left[f_i(k)\right]_k\right)$$

   (a) If $e_i(j) = e_i(f_i(j))$, then:

$$\forall j \in \left([0..k] - \bigcup_{q=1}^{n} O_{q,i}\right)\big(a\left[e_i(j)\right] = g^{j+1}\left(a\left[f_i(0)\right]\right)\big)$$

   (b) Otherwise,

$$\forall j \in \left([0..k] - \bigcup_{q=1}^{n} O_{q,i}\right)\big(a\left[e_i(j)\right] = g^{j+1}\left(a_0\left[f_i(j)\right]\right)\big)$$

2. For predicates of the form

$$a\left[e_i(k)\right]_{k+1} = g\left(a\left[f_i(k)\right]_{k+1}\right)$$

   drop the subscript $k+1$ and quantify as such:

$$\forall j \in \left([0..k] - \bigcup_{q=1}^{n} O_{q,i}\right)\left(a\left[e_i(j)\right] = g\left(a\left[f_i(j)\right]\right)\right)$$

This algorithm will be discussed in relation to the example of Program 5.

For this example, R_{k+1} is calculated and refined by (5.22) to give

$$R_{k+1} = (k \in \mathbb{N} \wedge a[k+1]_{k+1} = 2a[k]_k \wedge a[k]_{k+1} = 3 \wedge i_{k+1} = k+1 \wedge k < N)$$

There are two equalities in R_{k+1} derived from array assignments. Following the above naming conventions, $e_1(k) = k+1$, $e_2(k) = k$ and $f_1(k) = k$. The

Program 5 Array progression example

$\{\forall j \in [0..N]\,(a[j] = A[j]) \wedge i = 0 \wedge a[i] = 3\}$ The constant equality $N = \#a - 1$ also holds

```
do i<N →

    a[i+1]:=2*a[i];
    a[i]:=3;
    i:=i+1;

od;
```

second array assignment does not have a reference to a on the RHS, so there is no $f_2(k)$.

Applying (5.29) and (5.30), $O_{1,2} = \{\}$, $O_{2,1} = [0..k{-}1]$, $O_{1,1} = \{\}$ and $O_{2,2} = \{\}$. By algorithm 4, $POI = \{\}$.

The invariant properties of the assignment `a[i]:=3`, obtained by applying (5.31), form P_{as}:

$$P_{as} = \forall j \in [0..k]\,(a[j] = 3)$$

For the other assignment, heuristic algorithm 6 is applied. In this example, case 1(a) is matched and applied, because $e_1(j) = j + 1 = e_1(f_1(j)) = e_1(j)$. This yields the following invariant:

$$\forall j \in \left([0..k] - \bigcup_{q=1}^{2} O_{q,1}\right) \left(a\,[j+1] = 2^{j+1}\,(a\,[0])\right)$$

This simplifies to the predicate

$$\begin{aligned} & \forall j \in ([0..k] - [0..k-1]) \left(a\,[j+1] = 2^{j+1}\,(a\,[0])\right) \\ \equiv\ & a\,[k+1] = 2^{k+1}\,(a\,[0]) \end{aligned}$$

which is conjoined to P_{as} to give

$$\begin{aligned} P_{as} &\equiv \forall j \in [0..k]\,(a[k] = 3) \wedge a\,[k+1] = 2^{k+1}a[0] \\ &\equiv \forall j \in [0..k]\,(a[k] = 3) \wedge a\,[k+1] = 3\left(2^{k+1}\right) \end{aligned}$$

This describes the invariant array properties yielded from the assignments of the loop body.

The initial array properties that remain unchanged must also be defined. The *changed index set* for array assignment 1, calculated by applying (5.32) is

$$IC_1 = \{k+1\}$$

Similarly, the *changed index set* for array assignment 2, calculated by applying (5.32) is

$$IC_2 = [0..k]$$

Therefore, the changed index set for the loop, calculated by (5.33), is

$$\begin{aligned} IC &= IC_1 \cup IC_2 \\ &= [0..k+1] \end{aligned}$$

So, by (5.34), the *Unchanged Index Set* is calculated as

$$\begin{aligned} IU &= [0..\#a-1] - [0..k+1] \\ &= [0..N] - [0..k+1] \end{aligned}$$

The *Unchanged Array Invariant*, P_{UA}, is calculated by (5.35) to be

$$P_{UA} = \forall j \in ([0..N] - [0..k+1])\,(a[j] = a[j]_0)$$

Using the loop precondition, $(\forall j \in [0..N]\,(a[j] = A[j]) \wedge i = 0)$, P_{UA} can be simplified to

$$P_{UA} = \forall j \in ([0..N] - [0..k+1])\,(a[j] = A[j])$$

All predicates in R_{k+1}, yielded as a result of array assignments, are dropped from R_{k+1}. In the example, this leaves

$$R_{k+1} = (k \in \mathbb{N} \wedge i_{k+1} = k+1 \wedge k < N)$$

R_{k+1} is refined for the remaining predicates by generalising over k and dropping all variable subscripts, to give

$$P = (k \in \mathbb{N} \wedge i = k+1 \wedge k < N)$$

P_{as} and P_{UA} are conjoined to P to give a complete invariant, denoted P:

$$P \equiv \left(\begin{array}{l} k \in \mathbb{N} \wedge i = k+1 \wedge \\ \forall j \in [0..k]\,(a[j] = 3) \wedge a\,[k+1] = 3\left(2^{k+1}\right) \wedge \\ \forall j \in ([0..N] - [0..k+1])\,(a[j] = A[j]) \wedge k < N \end{array} \right)$$

Simplifying the invariant by (5.24) to remove references to the iteration count k, gives

$$P \equiv \left(\begin{array}{l} i > 0 \wedge \forall j \in [0..i-1]\,(a[j] = 3) \wedge a\,[i] = 3\left(2^{i}\right) \wedge \\ \forall j \in ([0..N] - [0..i])\,(a[j] = A[j]) \wedge i \leq N \end{array} \right)$$

Applying (5.26) the post condition is:

$$\begin{aligned} R &= \left(\left(\begin{array}{l} (N \leq i \wedge i = 0 \wedge \forall j \in [0..N]\,(a[j] = A[j]) \wedge a[i] = 3) \vee \\ \left(\begin{array}{l} i > 0 \wedge \forall j \in [0..i-1]\,(a[j] = 3) \wedge a\,[i] = 3\left(2^{i}\right) \wedge \\ \forall j \in ([0..N] - [0..i])\,(a[j] = A[j]) \wedge i = N \end{array} \right) \end{array} \right) \right) \\ &\equiv \left(\left(\begin{array}{l} (N \leq i \wedge i = 0 \wedge a[i] = 3) \vee \\ \left(i > 0 \wedge \forall j \in [0..N-1]\,(a[j] = 3) \wedge a\,[N] = 3\left(2^{N}\right) \wedge i = N\right) \end{array} \right) \right) \end{aligned}$$

Algorithm 7 Modified algorithm for calculating the invariant and postcondition for loops with array variable assignments.

For a loop of the form $\{Q\}$`do` $G \rightarrow S$ `od` , where S is non-branching:
Given an iterative-form invariant , R_{k+1}, such that $R_{k+1} = sp\,(Q_{k+1},\ S)$ apply the following steps:

1. Apply the rules in table 5.1 to solve all solvable difference equations in R_{k+1}. For any variable, $v \in \nu\,(R_{k+1})$, where v_0 is not known, report an uninitialised variable defect and use $v_0 = v_0$.
2. For each of these solutions, apply (5.22) to R_{k+1}.
3. Let there be $n \geq 0$ equalities $a\,[e_1(k)] = exp_1, ..., a\,[e_n(k)] = exp_n$ in R_{k+1}, where $a\,[e_i(k)] = exp_i$ represents the predicate yielded from the ith sequential array assignment in S.
4. For every p and q in $[1..n]$ calculate the overwrite set $O_{p,q}$ by (5.29).
5. For every p in $[1..n]$ calculate the overwrite set $O_{p,p}$ by (5.30).
6. Define the point of intersection overwrite set POI by applying Algorithm 4.
7. Create the array semantics invariant, P_{as}, by simplifying (5.31).
8. For all equalities in R_{k+1} of the form $a\,[e(k)] = a\,[f(k)]$, apply Algorithm 6 and conjoin the result to P_{as}
9. Calculate the *Unchanged Array Invariant*, P_{UA}, by (5.35)
10. Drop predicates, $a\,[e_p(k)] = exp_p$, for all $p \in [1..n]$ from R_{k+1}.
11. P is now equivalent to the remainder of R_{k+1} with all variable subscripts removed.
12. Conjoin P_{as} and P_{UA} to P to complete the invariant.
13. Simplify P.
14. Calculate the postcondition, R, by applying (5.26).

Notice how the quantified formula derived from P_{UA} becomes trivially true in the postcondition. This states that there are no elements of the array that were not changed by the body of the loop.

Algorithm 7 formalises the approach to deriving loop invariants and postconditions for all loops with non-branching loop bodies where all difference equations can be solved.

To demonstrate the application of algorithm 7, consider Program 6. Both assignments in this example are at some stage redundant.

Program 6 Array initialisation example with redundancy

The constant relation $N = \#a$ holds

```
i:=0; j:=N-1;
```

$\{i = 0 \wedge j = N - 1\}$

```
do i < j+4 →
    a[i]:=A;    (array assignment 1)
    a[j]:=B;    (array assignment 2)
    i:=i+1;
    j:=j+1;
od
```

1. The initial conditions on an arbitrary $k + 1$st iteration are:

$$Q_{k+1} = (k \in \mathbb{N} \wedge i_{k+1} = i_k \wedge j_{k+1} = j_k \wedge i_k < j_k + 4)$$

2. R_{k+1} is calculated to be:

$$R_{k+1} = \left(\begin{array}{l} k \in \mathbb{N} \wedge a\,[i_k] = A \wedge a\,[j_k] = B \wedge \\ i_{k+1} = i_k + 1 \wedge j_{k+1} = j_k - 1 \wedge i_k < j_k + 4 \end{array} \right)$$

3. The difference equations $i_{k+1} = i_k + 1$ and $j_{k+1} = j_k - 1$ are solved as $i(k) = k$ and $j(k) = N - k$ respectively.

4. By (5.22), with $i(k) = k$ and $j(k) = N - k$, R_{k+1} is refined to:

$$R_{k+1} = \left(\begin{array}{l} k \in \mathbb{N} \wedge a\,[k] = A \wedge a\,[N - k] = B \wedge \\ i_{k+1} = k + 1 \wedge j_{k+1} = N - k - 1 \wedge k < N - k + 4 \end{array} \right)$$

5. Array assignment 1, `a[i]:=A`, corresponds to the predicate $a[k] = A$ in R_{k+1}. This yields the array indexing function $e_1(k) = k$. Array assignment 2, `a[j]:=`B, corresponds to the predicate $a[N - k] = B$ in R_{k+1}. This yields the array indexing function $e_2(k) = N - k$.

6. Calculating the overwrite sets by (5.29):

$$\begin{aligned}
O_{2,1} &= \{j \in [0..k] \,|\, \exists l \in [j..k] \,(e_2(l) = e_1(j))\,\} \\
&\equiv \{j \in [0..k] \,|\, \exists l \in [j..k] \,(N - l = j)\,\} \\
&\equiv \{j \in [0..k] \,|\, j \leq N - j \leq k\,\} \\
O_{1,2} &= \{j \in [0..k] \,|\, \exists l \in [j..k] \,(e_1(l) = e_2(j))\,\} \\
&\equiv \{j \in [0..k] \,|\, \exists l \in [j..k] \,(l = N - j)\,\} \\
&\equiv \{j \in [0..k] \,|\, j \leq N - j \leq k\,\}
\end{aligned}$$

7. Calculating the overwrite sets by (5.30):

$$\begin{aligned}
O_{1,1} &= \{\} \\
O_{2,2} &= \{\}
\end{aligned}$$

8. The points of intersection of e_1 and e_2 are given by

$$\begin{aligned}
e_1(k) &= e_2(k) \\
k &= N - k \\
k &= \frac{N}{2}
\end{aligned}$$

Therefore, by applying algorithm (4) the point of intersection overwrite set is calculated as

$$POI = \left\{ \left(\frac{N}{2}, \frac{N}{2}, B \right) \right\}$$

Figure 5.2 provides a graphical representation of the two array indexing expressions as functions from the iteration constant to an array position. In this figure, $N = 20$ is assumed. The point of intersection at $\frac{N}{2}$ can easily be seen.

9. Applying (5.31) P_{as} is calculated as:

$$\begin{aligned}
P_{as} &\equiv \bigwedge_{p=1}^{n} \left(\begin{array}{l} \forall l \in \left([0..k] - \bigcup_{q=1}^{n} O_{q,p} \right) (a\,[e_p(l)] = exp_p) \wedge \\ \forall (it, x, exp) \in POI\,(it \leq k \Rightarrow a\,[x] = exp) \end{array} \right) \\
&\equiv \left(\begin{array}{l} \forall l \in ([0..k] - (O_{1,1} \cup O_{2,1}))\,(a\,[e_1(l)] = exp_1) \wedge \\ \forall l \in ([0..k] - (O_{1,2} \cup O_{2,2}))\,(a\,[e_2(l)] = exp_2) \wedge \\ \frac{N}{2} \leq k \Rightarrow a\left[\frac{N}{2}\right] = B \end{array} \right) \\
&\equiv \left(\begin{array}{l} \forall l \in ([0..k] - \{j \in [0..k] \,|\, j \leq N - j \leq k\})\,(a\,[l] = A) \wedge \\ \forall l \in ([0..k] - \{j \in [0..k] \,|\, j \leq N - j \leq k\})\,(a\,[N - l] = B) \wedge \\ \frac{N}{2} \leq k \Rightarrow a\left[\frac{N}{2}\right] = B \end{array} \right) \\
&\equiv \left(\begin{array}{l} \forall l \in \{j \in [0..k] \,|\, j > N - j \vee N - j > k\}\,(a\,[l] = A) \wedge \\ \forall l \in \{j \in [0..k] \,|\, j > N - j \vee N - j > k\}\,(a\,[N - l] = B) \wedge \\ \frac{N}{2} \leq k \Rightarrow a\left[\frac{N}{2}\right] = B \end{array} \right) \\
&\equiv \left(\begin{array}{l} \forall l \in \left\{j \in [0..k] \,\middle|\, j > \frac{N}{2} \vee N - j > k\right\}(a\,[l] = A \wedge a\,[N - l] = B) \wedge \\ \frac{N}{2} \leq k \Rightarrow a\left[\frac{N}{2}\right] = B \end{array} \right)
\end{aligned}$$

10. There are no equalities in R_{k+1} of the form $a\,[e(k)] = a\,[f(k)]$.

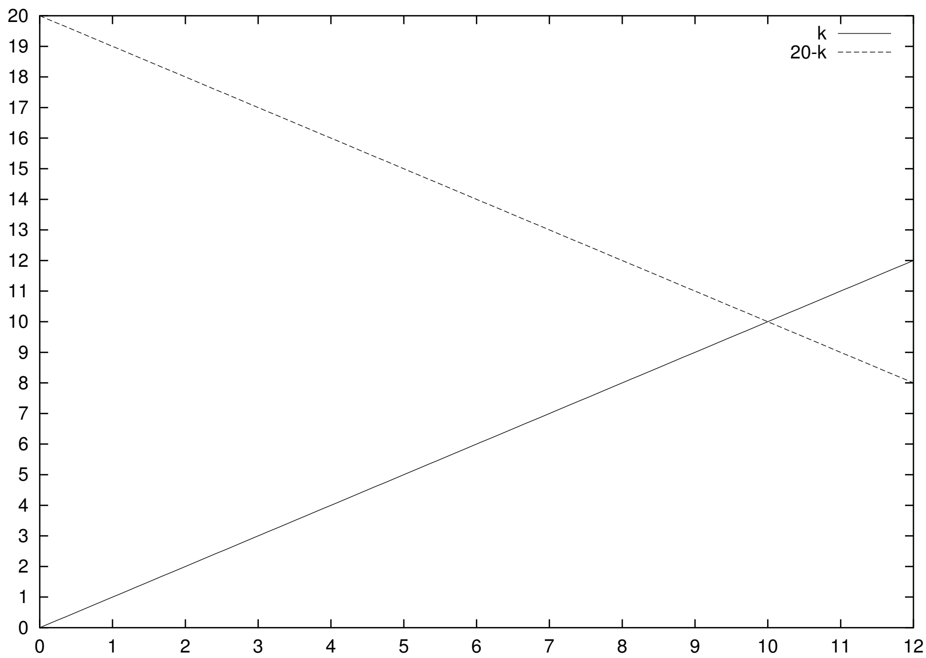


Figure 5.2: A graphical representation of Program 6 with $N = 20$

11. Applying ((5.35)) the unchanged array invariant is

$$P_{UA} = \forall j \in ([0..N-1] - ([N-k..N] \cup [0..k]))\,(a[j] = a[j]_0)$$

12. The array assignment predicates, $a\left[e_p(j)\right] = exp_p$, for all $p \in [1..n]$ are dropped from R_{k+1} leaving

$$R_{k+1} = (k \in \mathbb{N} \wedge i_{k+1} = k+1 \wedge j_{k+1} = N-k-1 \wedge k < N-k+4)$$

13. The remainder of R_{k+1} is refined by dropping the subscripts giving

$$P = (k \in \mathbb{N} \wedge i = k+1 \wedge j = N-k-1 \wedge k < N-k+4)$$

14. P_{as} and P_{UA} are conjoined to P giving:

$$P \quad \equiv \quad \left(\begin{array}{l} k \in \mathbb{N} \wedge i = k+1 \wedge j = N-k-1 \wedge k < N-k+4 \wedge \\ \forall l \in \left\{ j \in [0..k] \,\middle|\, j > \frac{N}{2} \vee N-j > k \right\} (a\,[l] = A \wedge a\,[N-l] = B) \wedge \\ \forall j \in [0..N-1] - ([N-k..N] \cup [0..k])\,(a[j] = a[j]_0) \wedge \\ \frac{N}{2} \le k \Rightarrow a\left[\frac{N}{2}\right] = B \end{array} \right)$$

15. P is simplified by applying (5.24) to give

$$\begin{array}{lll} P & \equiv & \left(\begin{array}{l} i > 0 \wedge j = N-i \wedge i-1 < N-i+1+4 \wedge \\ \forall l \in \left\{ j \in [0..i-1] \,\middle|\, j > \frac{N}{2} \vee N-j > i-1 \right\} (a\,[l] = A \wedge a\,[N-l] = B) \wedge \\ \forall j \in [0..\#N-1] - ([N-i+1..N] \cup [0..i-1])\,(a[j] = a[j]_0) \wedge \\ \frac{N}{2} \le i-1 \Rightarrow a\left[\frac{N}{2}\right] = B \end{array} \right) \\ & \equiv & \left(\begin{array}{l} i > 0 \wedge j = N-i \wedge i-1 < j+5 \wedge \\ \forall l \in \left\{ j \in [0..i-1] \,\middle|\, j > \frac{N}{2} \vee N-j \ge i \right\} (a\,[l] = A \wedge a\,[N-l] = B) \wedge \\ \forall j \in [0..N-1] - ([0..i-1] \cup [j+1..N])\,(a[j] = a[j]_0) \wedge \\ \frac{N}{2} \le i-1 \Rightarrow a\left[\frac{N}{2}\right] = B \end{array} \right) \end{array}$$

16. The postcondition is calculated by (5.26) as:

$$
\begin{array}{rcl}
R & = & (\neg G \wedge (Q \vee P)) \\
& \equiv & \left(\begin{array}{l} i \geq j+4 \wedge \\ \left(\begin{array}{l} i = 0 \wedge j = N \vee \\ \left(\begin{array}{l} i > 0 \wedge j = N - i \wedge i - 1 < j + 5 \wedge \\ \forall l \in \left\{ j \in [0..i-1] \,\middle|\, j > \frac{N}{2} \vee N - j \geq i \right\} (a\,[l] = A \wedge a\,[N-l] = B) \wedge \\ \forall j \in [0..N-1] - ([0..i-1] \cup [j+1..N])\,(a[j] = a[j]_0) \wedge \\ \frac{N}{2} \leq i - 1 \Rightarrow a\left[\frac{N}{2}\right] = B \end{array} \right) \end{array} \right) \end{array} \right)
\end{array}
$$

The first disjunct, $i \geq j+4 \wedge i = 0 \wedge j = N$, is $false$ because $N = \#a$ and so must be a natural number.

$$
\equiv \left(\begin{array}{l} i > 0 \wedge j = N - i \wedge j + 4 \leq i \leq j + 5 \wedge \\ \forall l \in \left\{ j \in [0..i-1] \,\middle|\, j > \frac{N}{2} \vee N - j \geq i \right\} (a\,[l] = A \wedge a\,[N-l] = B) \wedge \\ \forall j \in [0..N-1] - ([0..N])\,(a[j] = a[j]_0) \wedge \\ \frac{N}{2} \leq i - 1 \Rightarrow a\left[\frac{N}{2}\right] = B \end{array} \right)
$$

$$
\equiv \left(\begin{array}{l} i > 0 \wedge j = N - i \wedge j + 4 \leq i \leq j + 5 \wedge \\ \forall l \in \left\{ j \in [0..i-1] \,\middle|\, j > \frac{N}{2} \vee N - j \geq i \right\} (a\,[l] = A \wedge a\,[N-l] = B) \wedge \\ \frac{N}{2} \leq i - 1 \Rightarrow a\left[\frac{N}{2}\right] = B \end{array} \right)
$$

5.4.6 Detecting One-Too-Many and One-Too-Few Defects

It is possible to support defect detection by automating procedures to yield information that supports reasoning about a program in terms of certain classes of defect. This section describes a procedure to provide information to assist in investigating one-too-many/one-too-few type defects. An inspection rule is defined to make use of these procedures.

We can do a range analysis on all quantified expressions, by investigating what happens at the upper and lower bounds. Because these bounds are constructed from, k, the iteration count, they correspond to the initialisation and the last iteration of the loop. The corresponding analysis of the bounds of quantified expressions in a loop's postcondition can, therefore, be used to reveal defects where a loop body has been iterated one too many or one too few times. These are common defects that are usually the subject of specific tests, and generally only found in inspection by a symbolic execution of the code.

Program 7 Strange Array Initialisation Example

```
i := 0;
do i ≠ N →
    a[i]:= 0;
    i := i + 1;
    a[i] := 1;
    i := i + 2;
od
```

The postcondition of the example loop of Program 7 calculated by (5.26) is:

$$\begin{array}{rcl} R & = & (\neg G \wedge (Q \vee P)) \\ & \equiv & \left(\begin{array}{l} \frac{i}{3} \in \mathbb{N} \wedge i = N \wedge \\ \forall j \in \left[0..\frac{i}{3} - 1\right] (a\,[3j] = 0 \wedge a\,[3j+1] = 1) \end{array} \right) \end{array}$$

Investigating the quantified formula at its lower bound, $j = 0$, it is determine that $a[0]$ is the first element to be assigned a value of 0 and that $a[1]$ is the first element to be assigned a value of 1.

Investigating the upper bound, $j = \frac{i}{3} - 1 = \frac{N}{3} - 1$, of the quantified formula, it is determine that $a\left[3\left(\frac{N}{3} - 1\right)\right]$, that is $a\,[N-3]$ is the last element to be assigned a value of 0 and $a\,[N-2]$ is the last element to be assigned a value of 1.

This information is easily calculated for all quantified formula in a loop's postcondition, and has the potential to be provided automatically. Such a procedure is, therefore, useful as it removes the need for a verifier to perform static analysis or conduct testing, interpreting the results to determine the correctness of a loop with respect to the iteration count.

An Inspection Rule for Repeatable Array Bound Defect Detection

The procedure of investigation of array range references is summarised by inspection rule 3.

Inspection Rule 3. (Array Bounds Defect Detection)

For a loop of the form $\{Q\}$`do` $G \rightarrow S$ `od` , where S is non-branching, and the invariant after at least one iteration is P and the derived postcondition is R:

Investigate Array Reference Defects

For all quantified formulae in R of the form $Qj \in [l..u](e)$, where Q is a quantifier, derive the first element of the sequence:

$$first = e[l/j]$$

and derive the last element of the sequence:

$$last = e[u/j]$$

For any variable, v, where $v \in \nu(G) \cap \nu(e)$, for every equality, $v = E$, substitute $first[E/v]$, $last[E/v]$.

Investigate $first$ and $last$ for each substitution with respect to array bounds by answering the following inspection questions:

- Does $first$ refer to an array element less than zero?
- If the array size is bound, does $last$ refer to an array element beyond this upper bound of size?

- Does $first$ refer to an array element less than or greater than the first element expected to be described by the quantified formula (often 1 too small or one too large)?
- Does *last* refer to an array element less than or greater than the last element expected to be described by the quantified formula (often 1 too small or one too large)?

5.4.7 Related Work

Most closely related to the work in this section regarding the generation of invariants from program code with arrays is that of Katz and Manna [61]. In this work loop structures with certain properties must be matched with heuristic rules to yield an assertion about an array property maintained by a loop. These assertions are generally of the form $\forall j \in$ *<j-index>* (*<j-array>*) or $\exists j \in$ *<j-index>* (*<j-array>*), where *<j-index>* is a claim about array indices and *<j-array>* is a claim which is made about the array itself. They provide top-down heuristic rules for deriving invariant assertions from a known postcondition and loop information as well as bottom-up rules for constructing invariants from program code. The work presented in this section relates to the bottom-up rules.

Katz and Manna [61] present a number of rules for investigating loop structure to find the *<j-index>* and a number of rules for identifying the *<j-array>*. The rules for identifying the *<j-index>* are dependent on being able to find a counter variable in the loop body that is used as an array index and determine the set of values that that variable assumes when used as an array index. The rules for identifying the *<j-array>* rely on being able to determine if an assignment on an arbitrary iteration assigns a value to an element whose index is in the *<j-index>* at the commencement of that iteration and if the assertion yielded as a result of that assignment can be written as a conjunct which is valid for all elements added to the domain of *<j-index>*. Essentially these rules allow one to identify assertions yielded as a result of assignments in searches and sorts. It does not seem that they are applicable in the case of the poorly structured loops involving redundant assignments described in this chapter.

The algorithms presented in this chapter are capable of describing both the index properties and array assertions in a single calculation. The *<j-index>* is simply the range of the iteration count, which is always $[0..k]$ and the *<j-array>* assertion is presented with respect to a function of this range describing the set of indexed array elements that the assertion describes. This indexing function is obtained by simple substitution. So, for non-branching loops, if P is a predicate in R_{k+1} involving an array a, then a universally quantified invariant assertion of the form $\forall j \in [0..k](P[j/k])$ can always be calculated. If a variable exists for which there is a solvable difference equation with respect to k, then this variable may be used to substitute for k. This algorithmic approach lends itself a lot more to implementation than an entirely heuristic approach.

We present the following example, described in [61], to discuss the advantages and disadvantages of the method proposed in this thesis:

```
{a = a? ∧ a ∈ seq ℕ ∧ n < #a}
i:=0;
do i ≤ n →
    a[i]:=0;
    i := i + 7;
od
```

Applying the heuristic rules in [61], we would try to find a counter used as an index that is incremented or decremented by 1 in every iteration. As i ,the only index variable used, is not incremented or decremented by 1 we would then apply another heuristic rule to make an invariant assertion about i then use this assertion with a second heuristic to describe the <*j-index*>. Conditions on the equality resulting from the assignment are then matched and heuristic rules accordingly selected, eventually yielding the invariant assertion

$$\forall j\,(0 \leq j < i \wedge \exists n\,(n \geq 0 \wedge j = 7n) \Rightarrow a[j] = 0)$$

This approach relies on the application of a number of heuristic rules and does not assert anything about unchanged array elements.

The algorithms presented in this chapter have been largely automated. The prototype tool implementing Algorithm 7 is able to calculate the following invariant:

$$\left(\begin{array}{l} k \in \mathbf{N} \wedge n = n? \wedge 7 \times k \leq n \wedge i = 7 \times (1 + k) \wedge \\ \forall j \in [0..k]\,(a\,[7 \times j] = 0) \wedge \forall j \in ([0..\#a - 1] - \{e \in [0..k] \bullet 7 \times e\})\,(a\,[j] = a?\,[j]) \end{array}\right)$$

The implementation is straight forward as it is not reliant on matching conditions to select heuristics[5]. Notice that the derived invariant also asserts an invariant involving the array elements not changed as a result of the loop body.

The heuristic rules presented by Katz and Manna are capable of identifying invariant assertions from loops such as searches and sorts with a more complex loop body structure. The algorithms presented in this chapter are not capable of dealing with loops with branched loop bodies. We extend the approach presented here with heuristic rules in the next chapter to address these types of loops.

The boundary violation checking process and inspection rules in the previous section are comparable to a number of existing, automated static analysis techniques for identifying boundary violations such as those described in [5] and those implemented in the `lint` family of tools. The main benefit of the approach presented in this chapter is that it is integrated with the calculation of the postcondition. As such the prototype tool is capable of generating the boundary violation inspection questions as part of the invariant and postcondition calculation process.

[5]This is not the case where redundant assignments are involved.

5.5 Using Calculated Semantics for Maintenance

Consider again program 5 which yields the following postcondition:

$$R \equiv \left(\left(\begin{array}{l} N \leq i \wedge i = 0 \wedge \wedge a[i] = 3 \vee \\ i > 0 \wedge \forall j \in [0..N-1] \, (a[j] = 3) \wedge a\,[N] = 3\left(2^N\right) \wedge i = N \end{array} \right) \right)$$

This result highlights another use for the derived semantics. Assume that for this program, the calculated postcondition indicated that the program was correct. The loop above is actually an untidy and difficult to reason about algorithm for establishing this postcondition. Without the calculated postcondition, it requires quite a deal of thinking about to figure out what it is doing. For this reason it is likely to be difficult to maintain in the absence of a formal description. We can, however, use the calculated postcondition to derive a more readable and maintainable program, which also achieves this postcondition. For example, the above program can be rewritten as the more readable, and hence, maintainable:

$\{\forall j \in [0..N] \, (a[j] = A[j]) \wedge i = 0 \wedge a[i] = 3\}$ The constant equality $N = \#a - 1$ also holds

```
do i<N →
    a[i]:=3;
    i:=i+1;
od;
a[N]:=3× 2^N;
```

If this new program was not derived in a formal manner from the calculated postcondition, the postcondition would be calculated for this program and compared for correctness against that of the original. Following the steps outlined in this chapter, the postcondition of the loop in this new program is

$$\left(\begin{array}{l} N \leq i \wedge i = 0 \wedge \wedge a[i] = 3 \vee \\ i > 0 \wedge \forall j \in [0..N-1] \, (a[j] = 3) \wedge a\,[N] = A[N] \wedge i = N \end{array} \right)$$

Applying the strongest postcondition predicate transformer to this predicate and the assignment statement after the loop yields the following postcondition:

$$\left(\begin{array}{l} N \leq i \wedge i = 0 \wedge \wedge a[i] = 3 \vee \\ i > 0 \wedge \forall j \in [0..N-1] \, (a[j] = 3) \wedge a\,[N] = 3\left(2^N\right) \wedge i = N \end{array} \right)$$

Therefore, the new program is semantically equivalent to Program 5. However, the new program is more readable than the original. Automated tool support would make this a seamless verification step in the development and maintenance process.

5.6 Conclusion

In this chapter we have presented an approach to generate invariants and postconditions from the iterative-form invariant for loops with non-branching loop bodies. We discuss algorithms for generating invariants by solving difference equations in the iterative-form invariant. A table of difference equation solutions which are implemented in the current prototype is presented. A significant contribution of this chapter is a largely algorithmic approach for deriving universally quantified invariants from non-branching loops involving arrays. We compare this approach to the bottom-up heuristic approach in [61]. Further algorithms and heuristic rules are introduced which deal with deriving correct invariants from poorly structured loops involving redundant assignments to array elements. Inspection rules for finding boundary violations using the derived postcondition are also discussed.

Algorithm 7 is formulated describing a mechanisable process for generating all invariant assertions from non-branching loops. Of the steps in this algorithm, all except those dealing with redundant assignments (4-9) have been implemented in the prototype.

The algorithms discussed in this chapter are not capable of generating invariants for loops with branching loop bodies. The following chapter introduces algorithms and heuristics for dealing with many classes of loops fitting this description.

Chapter 6

MBS Loop Semantics

The derivation of invariants for loops with non-branching loop bodies, as defined in the previous chapter, is simply an application of mathematical induction. Loops with branched loop bodies are quite different and much more difficult to analyse. In this chapter we define a number of heuristic rules for deriving invariants from loops with multiple paths through the loop body. Before formally defining these rules, we discuss the problem of analysing the semantics of loops with multiple paths through the loop body.

6.1 Problems Deriving Invariants and Postconditions from Multiple-Path Loops

The strongest postcondition predicate transformer for loops [24] yields disjunctive predicates, where the number of disjuncts is bound by the number of potential iterations of the loop. For example, consider the Find-First-Maximum algorithm in Program 8.

Program 8 Find First Maximum[26] loop

$\{a[] = A[] \wedge N = \#a - 1\}$

```
i := 0; p := i;
do i ≠ N →
    if a[p] ≥ a[i+1] →
        i := i + 1;
    [] a[p] < a[i+1] →
        p := i+1;
        i := i + 1;
    fi;
od
```

The strongest postcondition of this program written as a disjunction is

$$\left(\begin{array}{l} p = 0 \wedge \forall j \in [1..N]\,(a[p] \geq a[j]) \vee \\ a[0] < a[p] \wedge p = 1 \wedge \forall j \in [2..N]\,(a[p] \geq a[j]) \vee \\ ...\vee \\ \forall j \in [0..p-1]\,(a[j] < a[p]) \wedge p = N \end{array}\right) \wedge i = N$$

The depth of this calculation is bound only by the arbitrary size of the sequence being searched. Due to its arbitrary size, the postcondition yielded is difficult to use in any form of verification, particularly verification based on human reasoning. A conjunctive form of the postcondition is generally much more useful for verification.

The problem of calculating a conjunctive invariant and postcondition, for a multiple-path loop, using only variables accessible to the loop is illustrated with the Find First Maximum loop. This loop has the *branch successor graph* (BSG) [80] shown in Fig. 6.1. Without knowing what a is initially, it is impossible

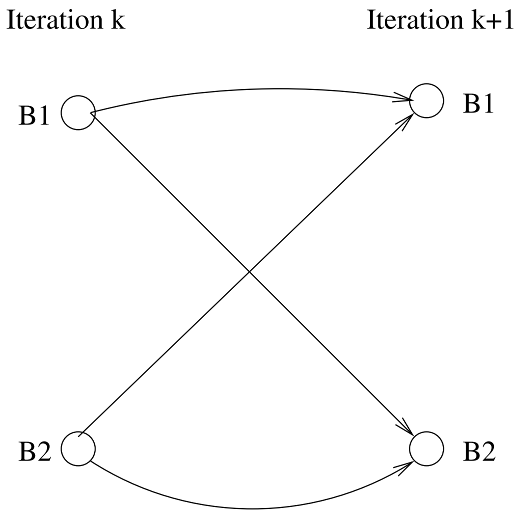


Figure 6.1: BSG for Find First Maximum

to determine how many times, and in what order, branches B_1 and B_2 will be iterated, and therefore, impossible to directly calculate a value for p with respect to a.

Formally this problem can be introduced by analysing the *update functions* of loop variables. The concept of an update function will be used to define the changes to a variable as a result of iterating a loop through a particular path.

Definition 7. For a loop `do G → S od`, where there are multiple paths through `S`, given any variable $x \in \nu(Q) \cup \nu(G) \cup \nu(S)$, $Q_{k+1} \Rightarrow x_{k+1} = x_k$ and $R_{i,k+1} \Rightarrow x_{k+1} = \phi_i(x_k)$, where $R_{i,k+1}$ is the disjunct of R_{k+1} corresponding to an iteration through path i. We call $\phi_i(x)$ the *update function* of x for path i.

For example, if $R_{i,k+1} \Rightarrow x_{k+1} = x_k + 1$, then the update function of x for path i, is given by $\phi_i(x) = x + 1$.

Any variable x accessable to a loop will have an update function corresponding to each of the different paths. In general, it is impossible to determine exactly how many times and in which order these update functions will be applied. A 2-path loop will be used to explain formally the problem of determining the succession of iterations.

Given any 2-path loop in which a variable x is updated by the function $\phi_1(x)$ in path 1 and $\phi_2(x)$ in path 2. The strongest invariant statement we can make about x after an arbitrary $k+1$st iteration is given by the following.

Let $(p_1, ..., p_n)$ be a combination satisfying

$$\exists n > 0\,(p_1 \geq 0 \wedge \forall j \in [2..n]\,(0 < p_j) \wedge p_1 + p_2 + ... + p_n = k + 1)$$

then

$$x_{k+1} = \phi_1^{p_1}\,(\phi_2^{p_2}\,(\phi_1^{p_3}\,(...\,(\phi_2^{p_n}\,(x_0)))))$$

Here we have introduced auxiliary variables$(p_1, \ldots, p_n)$ to construct a conjunctive form of invariant. Such a conjunctive form is difficult to use in reasoning about correctness as $p_1, \ldots, p_n$ are not accessible. Without direct execution under specific input conditions, values for $n, p_1, ..., p_n$ can not be determined. The problem is that unless n is known, and exact values for each of $p_1, ..., p_n$ are known, we can not, in general, describe the value of x and, therefore, can not describe the strongest invariant properties of x[1] .

Pan [80] suggests the derivation of the *Strongest Accessible Conjunctive Postcondition* and *Strongest Accessible Conjunctive Invariant* as a conjunctive form of postcondition and invariant, involving only variables accessible to the loop. The strongest accessible conjunctive postcondition and invariant derived must be implied by the strongest postcondition and invariant, respectively.

The remainder of this chapter focuses on calculation methods for deriving strongest accessible conjunctive invariants and postcondition directly from loop code.

6.1.1 Related Work

A number of techniques have been developed to proceed toward the automatic generation of invariants for loops with more than one path through the loop body [41, 62, 80, 9, 11, 36, 88, 30]. These techniques are largely focused on the automatic identification of candidate invariants useful in formal proofs of correctness, with the exception of [80] which focuses largely on the derivation of invariants to support *loop normalisation*.

[1] Under special conditions the strongest invariant properties of a variable x can, indeed, be derived from a disjunctive iterative-form invariant. The next section describes these special conditions, and presents an algorithm for calculating a value for x after a $k+1$st iteration where these conditions hold.

The work in this chapter is an extension of the algorithmic and heuristic techniques of [62] and [80]. The focus of the presented techniques is on the definition of heuristic algorithms, amenable to automation, and general in the sense that they are based on semantic transformations, not syntactic transformations, and are therefore language independent.

6.2 Constructing Invariants from MBS Loops

The remainder of this chapter defines an approach to constructing strong and conjunctive invariants for MBS loops by composing invariant properties which can be derived directly from the iterative-form invariant with invariants which are yielded as a result of transforming the iterative-form invariant. As such, we describe an approach which is not a single algorithm or technique, but a composition of techniques to calculate parts of the larger invariant, and rules for the composition of the parts. This approach is summarised in algorithm 8. Algorithm 8 is expanded at the end of this chapter to refer to the techniques discussed within for deriving parts of an invariant.

Algorithm 8 Invariant Construction Strategy

For an MBS loop $\texttt{do G} \rightarrow \texttt{if C}_1 \rightarrow \texttt{S}_1 \texttt{ [] } \ldots \texttt{[]C}_n \rightarrow \texttt{S}_n \texttt{ fi od}$,

1. Calculate R_{k+1} by the method of chapter 4.
2. Derive any invariant properties implied by R_{k+1}.
3. Calculate strong invariants from the branch-iterative invariants $R_{i,k+1}$
4. Infer invariant properties from these strong invariants
5. Compose the conjunctive loop invariant as the conjunction of the invariants implied by R_{k+1}, and the invariants inferred from the strong branch-invariants

6.2.1 Multiply Branched Statement (MBS) Form

In order to simplify the analysis of loops and the extraction of loop invariants by algorithm 8, we consider a normal form for loops called a *multiply branched statement* (MBS) form [28, 80]. Pan and Dromey [28] introduce a systematic technique called *loop rationalisation* to apply transformations to a loop, based on the *wp* and *sp* predicate transformers, to derive an equivalent, however simpler, MBS loop structure. A loop that has undergone transformation to the MBS form establishes the same postcondition as the original loop form.

An MBS loop has the form

$$\texttt{do G} \rightarrow \texttt{if C}_1 \rightarrow \texttt{S}_1 \texttt{ [] } \ldots \texttt{[]C}_n \rightarrow \texttt{S}_n \texttt{ fi od}$$

where $\texttt{C}_1 \vee ... \vee \texttt{C}_n \equiv \mathit{true}$ and $\forall i, j \in [1..n]\,(i \neq j \Rightarrow \texttt{C}_i \Rightarrow \neg\texttt{C}_j)$.

Definition 8. A *well-structured loop* is a loop which has a strongly connected branch successor graph (BSG). That is, every node, corresponding to a path through the loop body, connects directly to all other nodes [28].

In a well-structured loop there is no logical redundancy in the loop and branch guards.

Definition 9. A *normalised MBS loop* is defined as a well-structured loop with an MBS loop body.

We assume that all loops have been transformed, by the systematic approach defined in [28], to an equivalent, normalised MBS loop form, before application of algorithm 8.

Consider an MBS loop of the form:

$\{Q\}$
`do` $G \rightarrow$
  `if` $C_1 \rightarrow S_1$
  `[]` $C_2 \rightarrow S_2$
  ...
  `[]` $C_n \rightarrow S_n$
  `fi`
`od`

An iterative form invariant derived from such a loop has the following properties. Let $R_{i,k+1}$ be $sp\,(Q_{k+1} \wedge C_{k,i},\ S_i)$ for $i \in [1..n]$, then $R_{i,k+1}$ is conjunctive and calculable. $R_{i,k+1}$ is an iterative form-invariant for an iteration passing through branch i on an arbitrary $k+1$st iteration. The iterative form invariant for the entire loop, denoted by the usual R_{k+1}, is

$$R_{k+1} = \left(\bigvee_{i=1}^{n} R_{i,k+1} \right) \tag{6.1}$$

Consider program 8. The find first maximum loop's iterative-form invariant is calculated as

$$R_{k+1} \equiv \begin{array}{l} \left(k \in \mathbb{N} \wedge i_k \neq N \wedge a\,[p_k]_{k+1} \geq a\,[i_k+1]_{k+1} \wedge i_{k+1} = i_k + 1 \wedge p_{k+1} = p_k \wedge a_{k+1} = a_k\right) \vee \\ \left(k \in \mathbb{N} \wedge i_k \neq N \wedge a\,[p_k]_{k+1} < a\,[i_k+1]_{k+1} \wedge i_{k+1} = i_k + 1 \wedge p_{k+1} = i_k + 1 \wedge a_{k+1} = a_k\right) \end{array}$$

where

$$R_{1,k+1} \equiv \left(k \in \mathbb{N} \wedge i_k \neq N \wedge a\,[p_k]_{k+1} \geq a\,[i_k+1]_{k+1} \wedge i_{k+1} = i_k + 1 \wedge p_{k+1} = p_k \wedge a_{k+1} = a_k\right)$$

and

$$R_{2,k+1} \equiv \left(k \in \mathbb{N} \wedge i_k \neq N \wedge a\,[p_k]_{k+1} < a\,[i_k+1]_{k+1} \wedge i_{k+1} = i_k + 1 \wedge p_{k+1} = i_k + 1 \wedge a_{k+1} = a_k\right)$$

The properties of MBS loop are exploited in the remaining techniques for invariant derivation.

6.3 Deriving Conjunctive Invariants from the Iterative-Form Invariant

In this section we introduce theories and techniques to exploit the iterative-form invariant R_{k+1} to derive strong, conjunctive invariants for a loop. These techniques are applied to R_{k+1} as a whole as well as each of its disjuncts, $R_{i,k+1}$. Initially we introduce techniques for identifying simple relationships, explicitly maintained by all branch-iterative invariants. Techniques are then described to identify and derive formula describing conditional functions. Finally in this section, we describe an approach useful for deriving range and ordering type invariant relationships.

6.3.1 Common Formulae

A formula maintained by all branches must be invariant. That is, given an n-branch MBS loop yielding the iterative-form invariant $R_{k+1} \equiv \bigvee_{i\in[1..n]} R_{i,k+1}$, if $\forall i \in [1..n]\,(R_{i,k+1} \Rightarrow F)$, then F is invariant. If F describes a difference equation in iterative-form, then the solution to this difference equation, and any substitution in F using that solution, is also invariant.

For example, considering the loop in the Find First Maximum program, we can see from the iterative-form invariant calculated in section 6.2 that $R_{1,k+1} \Rightarrow (i_k \neq N \wedge i_{k+1} = i_k + 1)$ and $R_{2,k+1} \Rightarrow (i_k \neq N \wedge i_{k+1} = i_k + 1)$, so $R_{k+1} \Rightarrow (i_k \neq N \wedge i_{k+1} = i_k + 1)$, and both $i_k \neq N$ and $i_{k+1} = i_k + 1$ are invariant. The equality $i_{k+1} = i_k + 1$ describes an iterative-form difference equation, so the solution, $i(k) = k$ and the substitution of that solution in the common part of R_{k+1}, yielding $k \neq N \wedge i_{k+1} = k + 1$, are also invariant.

6.3.2 Conditional Functions

This section describes heuristic rules for solving iterative-form difference equations describing multiple update functions for scalar variables, where the iterative-form invariant has special properties. The properties that are considered here are those displayed in iterative-form invariants yielded from algorithms which calculate conditional sums, conditional products and conditional copies of sequences of values.

Conditional Sums

A conditional sum occurs in a program when a variable is used to sum the values of a sequence that meet a set of criteria. For example, the sum of the positive values in an array, or the sum of every third number between 0 and 100.

We begin by looking at the iterative-form of a 2-path conditional sum over the elements of a sequence. This form of conditional sum, will sum one of two expressions, depending on the outcome of some set of conditions. The set of conditions determining the value that is added to the sum is defined by the guard conditions for that path.

An iterative-form invariantR_{k+1}, that describes a 2-path conditional sum has the following property:

$$R_{k+1} \Rightarrow \left(\begin{array}{l} (v_{k+1} = v_k + E_1 \wedge C) \vee \\ (v_{k+1} = v_k + E_2 \wedge \neg C) \end{array} \right) \tag{6.2}$$

where v is a scalar variable, and E_1 and E_2 are expressions (possibly an implicit 0), which are composed of only constants, or variables associated with solvable difference equations independent of v. If any of the variables in either E_1 or E_2 are dependent on v then the following solution does not apply.

An iterative-form invariant with property (6.2), implies that after the corresponding loop has iterated k times ($k \geq 0$), the variable v will be equal to $v_0 + (k_1 \times E_1) + (k_2 \times E_2)$, where $k_1 \geq 0$ and $k_2 \geq 0$, and $k_1 + k_2 = k$. Presenting the information in this form is difficult and not very useful for verification. There is a need to introduce an iterative count, k_i, for every path through the loop. We realise that all of the k_1 iterations that add E_1 to v correspond to iterations where the guard condition C holds, and all k_2 iterations that add E_2 to v correspond to iterations where the guard condition $\neg C$ holds.

We can represent the value of v after some kth iteration, and hence the solution to the difference equation implied in (6.2), as

$$v(k) = v_0 + \sum_{j \in [0..k-1]} \left\{ \begin{array}{ll} E_1'[j/k], & \text{if } C'[j/k] \\ E_2'[j/k], & \text{if } \neg C'[j/k] \end{array} \right. \tag{6.3}$$

where E_1', E_2' and C' are the expressions E_1, E_2 and C, respectively, with all variables substituted with their respective difference equation solutions. These solutions refer to the iterative count k. We use a variable, j that is not used by the program, as a bound variable for the sum to generalise the iterative-form sum over all iterations between iteration 0 and $k-1$.

Consider the following loop, for example:

```
{s ∈ seq T ∧ #s = N}
i := 0; sum := 0;
do i < N →
    if s[i] ≥ 0 →
        sum := sum + s[i]; i := i + 1;
    [] s[i] < 0 →
        sum := sum + -2s[i]; i := i + 1;
    fi
od
```

Calculating the iterative form invariant, we get

$$R_{k+1} \equiv \left[\begin{array}{c} k \in \mathbb{N} \wedge i_{k+1} = i_k + 1 \wedge i_k < N \wedge s_{k+1} = s_k \wedge \\ \left(\begin{array}{l} (sum_{k+1} = sum_k + s\,[i_k] \wedge s\,[i_k] \geq 0) \vee \\ (sum_{k+1} = sum_k + -2s\,[i_k] \wedge s\,[i_k] < 0) \end{array} \right) \end{array} \right]$$

We first solve the common difference equations yielding the solution $i(k) = k$ for i, where $i_0 = 0$.

R_{k+1} implies

$((sum_{k+1} = sum_k + s\,[i_k] \wedge s\,[i_k] \geq 0) \vee (sum_{k+1} = sum_k + -2s\,[i_k] \wedge s\,[i_k] < 0))$

By (6.2), we determine that R_{k+1} implies a 2-path conditional sum. By (6.3), we calculate the difference equation solution for sum as

$$sum(k) = \sum_{j \in [0..k-1]} \begin{cases} s\,[j]\,, & \text{if } s[j] \geq 0 \\ -2s\,[j]\,, & \text{if } s[j] < 0 \end{cases}$$

We now have difference equation solutions for all variables accessible to the loop and can compose a conjunctive form invariant, P. Referring to the solutions for i and sum we can state that after an arbitrary $k+1$st iteration, $i = k+1$ and $sum = \sum_{j \in [0..k]} \begin{cases} s\,[j]\,, & \text{if } s[j] \geq 0 \\ -2s\,[j]\,, & \text{if } s[j] < 0 \end{cases}$. As this has been determined for an arbitrary $k \geq 0$, we can compose the following invariant, P:

$$\begin{aligned} P &\equiv \left(k \in \mathbb{N} \wedge i = k+1 \wedge k < N \wedge sum = \sum_{j \in [0..k]} \begin{cases} s\,[j]\,, & \text{if } s[j] \geq 0 \\ -2s\,[j]\,, & \text{if } s[j] < 0 \end{cases} \right) \\ &\equiv \left(1 \leq i \leq N \wedge sum = \sum_{j \in [0..i-1]} \begin{cases} s\,[j]\,, & \text{if } s[j] \geq 0 \\ -2s\,[j]\,, & \text{if } s[j] < 0 \end{cases} \right) \end{aligned}$$

The above two branch refinement rule can be extended for an n branch conditional sum. An iterative-form invariant calculating an n-branch conditional sum with a variable v, has the property

$$R_{k+1} \Rightarrow \ldots \wedge \left(\begin{array}{l} v_{k+1} = v_k + E_1 \wedge C_1 \vee \\ v_{k+1} = v_k + E_2 \wedge C_2 \vee \\ \ldots \\ v_{k+1} = v_k + E_n \wedge C_n \end{array} \right) \tag{6.4}$$

where v is a scalar variable, $E_1, \ldots, E_n$ are expressions composed of only constants, or variables associated with solvable difference equations whose solutions are independent of v, and $C_1, \ldots, C_n$ are conditions corresponding to guard conditions for the n respective paths.

Extending (6.3), we can establish the following definition for the difference equation solution for v:

$$v(k) = v_0 + \sum_{j \in [0..k-1]} \begin{cases} E_1'[j/k], & \text{if } C_1{}'[j/k] \\ E_2'[j/k], & \text{if } C_2{}'[j/k] \\ \ldots \\ E_n'[j/k], & \text{if } C_n{}'[j/k] \end{cases} \tag{6.5}$$

where $E'_1, \ldots, E'_n$ and $C'_1, \ldots, C'_n$ are the expressions $E_1, \ldots, E_2$ and $C_1, \ldots, C_n$, respectively, with all variables substituted with their respective difference equation solutions.

Conditional Enumeration The term conditional enumeration is used to describe a class of program that counts only items of a sequence which meet some selection criteria. For example, a program to count the number of even numbers in a sequence would be an example of a conditional enumeration with the selection criteria being that the items counted are even numbers.

Conditional enumerations must increment some variable, say *count*, by one for every element in the sequence that matches the selection criteria C. An n-path conditional enumeration will yield an iterative-form invariant with the following properties:

$$R_{k+1} \Rightarrow \ldots \vee (count_{k+1} = count_k + 1 \wedge C_a) \vee \ldots \vee (count_{k+1} = count_k \wedge C_b) \vee \ldots \tag{6.6}$$

where $C_1, \ldots, C_n$ are conditions and $a, b \in [1..n]$, and *count* is a natural number, and at least one branch increments *count* by 1 and at least one branch does not update *count*, and all branches either do not update *count* or increment *count* by 1.

Because some of the paths do not increment count, there is no single difference equation to solve for *count*. The iterative form implied by (6.6) is, however, just a special case of that implied (6.4) where the E_is are either the constant 1 or 0.

If we let U be the set of all path conditions, C'_i, where $R_{i,k+1} \Rightarrow count_{k+1} = count_k + 1 \wedge C_i$, and C'_i is the condition C_i with all difference equation solutions substituted, then from (6.6) we can deduce the following difference equation solution for *count*

$$count(k) = count_0 + \sum_{j \in [0..k-1]} \begin{cases} 1, & \text{if } \bigvee_{c \in U} c[j/k] \\ 0, & \text{otherwise} \end{cases} \tag{6.7}$$

This can be written more naturally using set notation as:

$$count(k) = count_0 + \# \left\{ j \in [0..k-1] \mid \bigvee_{c \in U} c[j/k] \right\} \tag{6.8}$$

For example, consider the following simple loop to count the positive numbers in an array of numbers:

$\{s \in \mathtt{seq}\ \mathbb{Z} \wedge \#s = N\}$
```
i := 0; count := 0;
do i < N →
    if s[i] ≥ 0 →
```

```
            count := count + 1;
            i := i + 1;
        [] s[i] < 0 →
            i := i + 1;
        fi
    od
```

Calculating the iterative form invariant, we get

$$R_{k+1} \quad \equiv \quad \left[\begin{array}{l} k \in \mathbb{N} \wedge i_{k+1} = i_k + 1 \wedge i_k < N \wedge \\ ((count_{k+1} = count_k + 1 \wedge s\,[i_k] \geq 0) \vee (count_{k+1} = count_k \wedge s\,[i_k] < 0)) \end{array} \right]$$

We first solve the iterative-form difference equations common to all disjuncts of R_{k+1}. In this case, the iterative-form difference equation $i_{k+1} = i_k + 1$, where $i_0 = 0$ has the solution $i(k) = k$.

Examining R_{k+1}, we see that

$$R_{k+1} \Rightarrow ((count_{k+1} = count_k + 1 \wedge s\,[i_k] \geq 0) \vee (count_{k+1} = count_k \wedge s\,[i_k] < 0))$$

This has the properties described by (6.6). We can, therefore, calculate the following difference equation solution for *count* by (6.7):

$$count(k) = \sum_{j \in [0..k-1]} \left\{ \begin{array}{ll} 1, & \text{if } s[i(j)] \geq 0 \\ 0, & \text{otherwise} \end{array} \right.$$

By (6.8), this is equivalent to the more natural form using set notation

$$count(k) = \#\,\{j \in [0..k-1] \mid s[i(j)] \geq 0\}$$

Substituting the difference equation solution for i, this is simplified to

$$count(k) = \#\,\{j \in [0..k-1] \mid s[j] \geq 0\}$$

As there are no other variables involved we can produce the invariant P by substitution and composition.

$$P \equiv (k \in \mathbb{N} \wedge i = k + 1 \wedge k < N \wedge count = \#\,\{j \in [0..k] \mid s[j] \geq 0\})$$

Removing all references to the iteration count k, by substitution, this invariant is simplified to

$$P \equiv (1 \leq i \leq N \wedge count = \#\,\{j \in [0..i-1] \mid s[j] \geq 0\})$$

Using this invariant, we then apply (5.26) to calculate the postcondition

$$R \equiv (i = N \wedge count = \#\,\{j \in [0..i-1] \mid s[j] \geq 0\})$$

Conditional Products

Similar reasoning to that above may be used to derive results for condition products.

A conditional product occurs in a program when a variable is used to represent the product of those elements of a sequence meeting a set of criteria.

An iterative form invariant, R_{k+1}, of an n-path MBS loop which specifies a conditional product will have the following property:

$$R_{k+1} \Rightarrow \ldots \wedge \left(\begin{array}{l} v_{k+1} = v_k \times E_1 \wedge C_1 \vee \\ v_{k+1} = v_k \times E_2 \wedge C_2 \vee \\ \ldots \\ v_{k+1} = v_k \times E_n \wedge C_n \end{array} \right) \tag{6.9}$$

where v is a scalar variable, $E_1, \ldots, E_n$ are expressions composed of only constants, or variables associated with solvable difference equations whose solutions are independent of v, and $C_1, \ldots, C_n$ are conditions corresponding to guard conditions for the n respective paths.

From this iterative-form we can establish the following definition for the difference equation solution for v:

$$v(k) = v_0 \times \prod_{j \in [0..k-1]} \begin{cases} E_1'[j/k], & \text{if } C_1{}'[j/k] \\ E_2'[j/k], & \text{if } C_2{}'[j/k] \\ \ldots \\ E_n'[j/k], & \text{if } C_n{}'[j/k] \end{cases} \tag{6.10}$$

where $E_1', \ldots, E_n'$ and $C_1', \ldots, C_n'$ are the expressions $E_1, \ldots, E_2$ and $C_1, \ldots, C_n$, respectively, with all variables substituted with their respective difference equation solutions.

For example, consider the following loop:

```
{s ∈ seq T ∧ #s = N}
i := 0; prod := 1;
do i < N →
    if s[i] > 0 →
        prod := prod * s[i]; i := i + 1;
    [] s[i] ≤ 0 →
        prod := prod * ((s[i]+1)²); i := i + 1;
    fi
od
```

Calculating the iterative form invariant, we get

$$R_{k+1} \equiv \left[\begin{array}{l} k \in \mathbb{N} \wedge i_{k+1} = i_k + 1 \wedge i_k < N \wedge s_{k+1} = s_k \wedge \\ \left(\begin{array}{l} (prod_{k+1} = prod_k s\,[i_k] \wedge s\,[i_k] > 0) \vee \\ \left(prod_{k+1} = prod_k \left((s\,[i_k] + 1)^2\right) \wedge s\,[i_k] \leq 0\right) \end{array} \right) \end{array} \right]$$

We first solve the common difference equations yielding the solution $i(k) = k$ for i, where $i_0 = 0$.

The formula

$$\left((prod_{k+1} = prod_k s\,[i_k] \wedge s\,[i_k] > 0) \vee \left(prod_{k+1} = prod_k \left((s\,[i_k] + 1)^2\right) \wedge s\,[i_k] \leq 0\right)\right)$$

implied by R_{k+1}, indicates by comparison with the form of (6.9), that R_{k+1} describes a 2-path conditional product.

By (6.10), we calculate the difference equation solution for $prod$ as

$$prod(k) = \prod_{j \in [0..k-1]} \begin{cases} s\,[j]\,, & \text{if } s[j] > 0 \\ (s\,[j] + 1)^2\,, & \text{if } s[j] \leq 0 \end{cases}$$

We now have difference equation solutions for all variables accessible to the loop and can compose a conjunctive form invariant, P.

$$\begin{aligned} P \quad &\equiv \quad \left(k \in \mathbb{N} \wedge i = k + 1 \wedge k < N \wedge prod = \prod_{j \in [0..k]} \begin{cases} s\,[j]\,, & \text{if } s[j] > 0 \\ (s\,[j] + 1)^2\,, & \text{if } s[j] \leq 0 \end{cases}\right) \\ &\equiv \quad \left(1 \leq i \leq N \wedge prod = \prod_{j \in [0..i-1]} \begin{cases} s\,[j]\,, & \text{if } s[j] > 0 \\ (s\,[j] + 1)^2\,, & \text{if } s[j] \leq 0 \end{cases}\right) \end{aligned}$$

Rules for Universally Quantified Conditional Formulae

This section defines heuristic refinement rules for deriving universally quantified formulae from disjunctive iterative-form invariants yielded from MBS loops. We attempt to match the structure of the iterative-form invariant to a form which indicates a quantified conditional formula.

Assume that we calculate R_{k+1} for an n-path loop, where some predicate, F, regarding an array variable, a, is maintained by every path, through the loop body, then $R_{k+1} \Rightarrow F(a)$ would hold. By the procedure presented in the previous chapter, if $F(a)$ does not involve an equality of the form $a\,[f\,(i_k)]_{k+1} = g\,(a\,[f\,(i_k)]_k)$, we could generalise this iterative form over all iterations and state that $\forall j \in [0..k]\,(F'(a)[j/k])$ is invariant, where F' is F with all variables substituted with their respective difference equation solutions.

Now, consider a multi-path loop which maintains a predicate of the above form for at least one, but not all paths through the loop. We can no longer generalise as we did above because $F(a)$ is dependent on some set of conditions being met. These conditions correspond to the disjunction of the guards for the paths which imply that $F(a)$ holds. We can strengthen the above universally quantified formula to a form

$$\forall j \in [0..k]\,\left(\text{conditions met on iteration} \Rightarrow F'(a)[j/k]\right)$$

It follows from this discussion that the following theorem holds.

Theorem 6.1. [Conditional Universal Quantification]

For a sequence variable, a: If $\neg(R_{k+1} \Rightarrow (F(a)))$, and $\exists p \in [1..n] (R_{p,k+1} \Rightarrow (F(a)))$, and U is the set of all, C'_p, where C_p is the path condition for a path, p, where $R_{p,k+1} \Rightarrow (F(a))$, and C'_p is C_p with all variables substituted with difference equation solutions, then we can generalise the following:

$$\forall j \in [0..k] \left(\left(\bigvee_{c \in U} c[j/k] \right) \Rightarrow F'(a)[j/k] \right) \tag{6.11}$$

where $F'(a)$ is $F(a)$ with all variables substituted with their respective difference equation solutions.

Consider, for example, the following simple branched loop to copy the positive values from array a to array b.

```
i := 0; j:=0;
do i != N →
    if a[i] < 0 →
        i:=i+1;
    [] a[i] >= 0 →
        b[j] := a[i];
        i:=i+1; j:=j+1;
    fi;
od;
```

We are given no precondition, so we assume the precondition *true* for this code segment. This means that the loop precondition, by applying the strongest post-condition predicate transformer to the initialisation, would be $Q = \{i = 0 \wedge j = 0\}$.

The iterative-form invariant is calculated as

$$R_{k+1} \equiv \left[\begin{array}{l} k \in \mathbb{N} \wedge i_{k+1} = i_k + 1 \wedge i_k \neq N \wedge \\ \left(\begin{array}{l} j_{k+1} = j_k \wedge a\left[i_k\right]_{k+1} < 0 \vee \\ j_{k+1} = j_k + 1 \wedge a\left[i_k\right]_{k+1} \geq 0 \wedge b\left[j_k\right]_{k+1} = a\left[i_k\right]_{k+1} \end{array} \right) \end{array} \right]$$

Splitting this invariant to examine the two branch-iterative invariants we find

$$R_{1,k+1} \equiv \left(k \in \mathbb{N} \wedge i_{k+1} = i_k + 1 \wedge j_{k+1} = j_k \wedge a\left[i_k\right]_{k+1} < 0 \wedge i_k \neq N\right)$$

from path 1 and from path 2

$$R_{2,k+1} \equiv \left(\begin{array}{l} k \in \mathbb{N} \wedge i_{k+1} = i_k + 1 \wedge j_{k+1} = j_k + 1 \wedge \\ a\left[i_k\right]_{k+1} \geq 0 \wedge b\left[j_k\right]_{k+1} = a\left[i_k\right]_{k+1} \wedge i_k \neq N \end{array} \right)$$

We first solve the difference equations for all scalar variables in R_{k+1}.

- The iterative-form difference equation $i_{k+1} = i_k + 1$ is common to all branch-iterative invariants. This iterative form difference equation, where $i_0 = 0$, has the solution $i(k) = k$.
- We can see that

  $$R_{k+1} \Rightarrow \left(j_{k+1} = j_k \wedge a\left[i_k\right]_{k+1} < 0\right) \vee \left(j_{k+1} = j_k + 1 \wedge a\left[i_k\right]_{k+1} \geq 0\right).$$

  This information is of the form described by (6.6), where j is the *count* variable. By (6.7), where $j_0 = 0$, we establish

  $$j(k) = \#\left\{e \in [0..k-1] \mid a[i(e)] \geq 0\right\}$$

  From the above solution, we know that $i(e) = e$, so substituting we get

  $$j(k) = \#\left\{e \in [0..k-1] \mid a[e] \geq 0\right\}$$

Next we apply theorem 6.1 to universally quantify the relationship between $b\left[j_k\right]_{k+1}$ and $a\left[i_k\right]_{k+1}$. Applying this theorem, where $U = \{a[k] \geq 0\}$ and $P(a) = b\left[j_k\right]_{k+1} = a\left[i_k\right]_{k+1}$, and hence $P'(a) = b\left[\#\left\{e \in [0..k-1] \mid a[i(e)] \geq 0\right\}\right]_{k+1} = a\left[k\right]_{k+1}$, we obtain the invariant formula

$$\begin{aligned} & \forall l \in [0..k]\left((a[k] \geq 0)[l/k] \Rightarrow \left(b\left[\#\left\{e \in [0..k-1] \mid a[e] \geq 0\right\}\right] = a[k]\right)[l/k]\right) \\ \equiv\ & \forall l \in [0..k]\left(a[l] \geq 0 \Rightarrow b\left[\#\left\{e \in [0..l-1] \mid a[e] \geq 0\right\}\right] = a[l]\right) \end{aligned}$$

By applying the rules so far given, we have established the following invariant properties, P, from the syntax of this loop.

$$P \equiv \left[\begin{array}{l} k \in \mathbb{N} \wedge i = k+1 \wedge k \neq N \wedge \\ j = \#\left\{e \in [0..k] \mid a[e] \geq 0\right\} \wedge \\ \forall l \in [0..k]\left(a[l] \geq 0 \Rightarrow b\left[\#\left\{e \in [0..l-1] \mid a[e] \geq 0\right\}\right] = a[l]\right) \end{array}\right]$$

We can remove k from P by substituting equals for equals, to give

$$P \equiv \left[\begin{array}{l} i > 0 \wedge i-1 \neq N \wedge \\ j = \#\left\{e \in [0..i-1] \mid a[e] \geq 0\right\} \wedge \\ \forall l \in [0..i-1]\left(a[l] \geq 0 \Rightarrow b\left[\#\left\{e \in [0..l-1] \mid a[e] \geq 0\right\}\right] = a[l]\right) \end{array}\right]$$

6.3.3 Deriving Range Invariants

The solutions and refinements of the above sections can only be applied when the iterative-form invariants, yielded from their corresponding loops, have certain properties. They are, therefore, limited to special classes of loop. That is, in general, for a variable x, updated by differing update functions in different paths, calculation of the value of x after some $k+1$st iteration may not be possible. Given this problem, we attempt to derive weaker invariant properties of x by describing the constraints on the value of x. In the next two sections, we demonstrate techniques for deriving such invariant constraints.

For example, consider the program:

```
x := 0;
do G →
    if C →
        x := x + 2;
    [] ¬C →
        x := x * 4;
    fi
od
```

By the rules of the above section, we are not able to calculate a single solution for x after a $k{+}1$st iteration as the yielded iterative form invariant does not have any of the required properties. However, we can apply techniques discussed in the following sections to derive the following invariant properties of x:

$$k \in \mathbb{N} \wedge 0 \leq x \leq 2 \times 4^k \wedge \frac{x}{2} \in \mathbb{N}$$

Minimum and Maximum Calculation Rules for Numeric Variable Assignment

In this section we define heuristic rules for calculating the minimum and maximum values for a variable, x, with respect to the iteration count k, in an n-branch MBS loop. The minimum and maximum for x are denoted $min(x)$ and $max(x)$, respectively, and the following invariant can be stated:

$$min(x) \leq x \leq max(x) \tag{6.12}$$

We examining a number of special cases, occurring commonly in programs, for which the implication of range is easy to mechanically recognise by pattern matching.

Special Cases for MBS loops

For any variable x, if $\phi_1(x) = x + \alpha_1$, $\phi_2(x) = x + \alpha_2$, $\ldots$, $\phi_n(x) = x + \alpha_n$, and $\alpha_1, \ldots, \alpha_n$ are all constants or variables not updated by the loop and $\forall i \in [1..n]\,(\alpha_i \geq 0)$, then $min(x)$ and $max(x)$ are given by

$$\begin{aligned} min(x) &= x_i(k+1), \text{ where } \alpha_i = min\,(\alpha_1, \ldots, \alpha_n) \\ max(x) &= x_i(k+1), \text{ where } \alpha_i = max\,(\alpha_1, \ldots, \alpha_n) \end{aligned} \tag{6.13}$$

For any variable x, if $\phi_1(x) = \alpha_1 x$, $\phi_2(x) = \alpha_2 x$, $\ldots$, $\phi_n(x) = \alpha_n x$, and $\alpha_1, \ldots, \alpha_n$ are all constants or variables not updated by the loop and $\forall i \in [1..n]\,(\alpha_i \geq 1)$, then $min(x)$ and $max(x)$ are given by

$$\begin{aligned} min(x) &= x_i(k+1), \text{ where } \alpha_i = min\,(\alpha_1, \ldots, \alpha_n) \\ max(x) &= x_i(k+1), \text{ where } \alpha_i = max\,(\alpha_1, \ldots, \alpha_n) \end{aligned} \tag{6.14}$$

For any variable x, if $\phi_1(x) = x - \alpha_1$, $\phi_2(x) = x - \alpha_2$, ..., $\phi_n(x) = x - \alpha_n$, and $\alpha_1, \ldots, \alpha_n$ are all constants or variables not updated by the loop and $\forall i \in [1..n]\,(\alpha_i \geq 0)$, then $min(x)$ and $max(x)$ are given by

$$\begin{aligned} min(x) &= x_i(k+1), \text{ where } \alpha_i = max\,(\alpha_1, \ldots, \alpha_n) \\ max(x) &= x_i(k+1), \text{ where } \alpha_i = min\,(\alpha_1, \ldots, \alpha_n) \end{aligned} \tag{6.15}$$

For any variable x, if $\phi_1(x) = \frac{x}{\alpha_1}$, $\phi_2(x) = \frac{x}{\alpha_2}$, ..., $\phi_n(x) = \frac{x}{\alpha_n}$, and $\alpha_1, \ldots, \alpha_n$ are all constants or variables not updated by the loop and $\forall i \in [1..n]\,(\alpha_i \geq 1)$, then $min(x)$ and $max(x)$ are given by

$$\begin{aligned} min(x) &= x_i(k+1), \text{ where } \alpha_i = max\,(\alpha_1, \ldots, \alpha_n) \\ max(x) &= x_i(k+1), \text{ where } \alpha_i = min\,(\alpha_1, \ldots, \alpha_n) \end{aligned} \tag{6.16}$$

These rules, while restrictive, are often all that is required for deriving invariants relating to variable ranges. This is particularly so when dealing with loops that manipulate sequences.

Consider again, the find first maximum example. From the iterative form invariant we see that $R_{1,k+1} \Rightarrow p_{k+1} = p_k$, hence $\phi_1(p) = p$, that is, ϕ_1 is the identity function. $R_{2,k+1} \Rightarrow p_{k+1} = p_k + 1$, so $\phi_2(p) = p + 1$. We can apply rule (6.13), with $\alpha_1 = 0$ and $\alpha_2 = 1$. The corresponding difference equation solutions are $p_1(k) = p_0 = 0$ and $p_2(k) = k$.

$$\begin{aligned} min(p) &= p_i(k+1), \text{ where } \alpha_i = min\,(\alpha_1, \ldots, \alpha_n) \\ &= p_1(k+1), \text{ because } \alpha_1 < \alpha_2 \\ &= 0 \\ max(p) &= p_i(k+1), \text{ where } \alpha_i = max\,(\alpha_1, \ldots, \alpha_n) \\ &= p_2(k+1), \text{ because } \alpha_1 < \alpha_2 \\ &= k+1 \end{aligned}$$

Therefore, by (6.12), we can state the invariant

$$0 \leq p \leq k+1$$

after an arbitrary $k+1$st iteration, for any $k \geq 0$.

Special Addition-Multiplication Rule

The following lemmas introduce the special rules to find the minimum and maximum for a 2-branch MBS loops where one branch updates x by a sum and the other by a product.

Lemma 6.1. If $\phi_1(x) = x + \alpha$ and $\phi_2(x) = \beta x$ and $0 \leq x_0 < \alpha$ and $\alpha > 0$, and $\beta > 1$ and $\beta x_0 < x_0 + \alpha$, then then $min(x)$ and $max(x)$ are given, respectively, by

$$\begin{aligned} min(x) &= \beta x_0 \times (\alpha k) \\ max(x) &= (x_0 + \alpha) \times \beta^k \end{aligned} \tag{6.17}$$

Proof. We know that $\beta > 1$, so after the first iteration, applying $\phi_2(x)$ to the value of x, will yield a greater value for x, than will applying $\phi_1(x)$, because $(x_0 + \alpha)\beta \geq (x_0 + \alpha + \alpha)$, will always hold. Because, $\beta x_0 < x_0 + \alpha$, then the first iteration will yield the maximum value for x, when $\phi_1(x)$ is applied. □

Similarly, if $0 < x_0$ and $\phi_1(x) = x + \alpha$ and $\phi_2(x) = \beta x$ and $\alpha < \beta$, and $\alpha > 0$, and $\beta > 1$, then $min(x)$ and $max(x)$ are given, respectively, by

$$\begin{aligned} min(x) &= x_0 + \alpha(k+1) \\ max(x) &= x_0 \times \beta^{k+1} \end{aligned} \tag{6.18}$$

Applying (6.17) to the example in the introduction of this section, we have $x_0 = 0$, $\phi_1(x) = x + 2$, $\phi_2(x) = 4x$, so $\alpha = 2$ and $\beta = 4$. We, see that the conditions $0 \leq x_0 < \alpha$, $\alpha > 0$, $\beta > 1$ and $\beta x_0 < x_0 + \alpha$ all hold, and so we obtain

$$\begin{aligned} min(x) &= \beta x_0 \times (\alpha k) \\ &= 0 \times 2k \\ &= 0 \\ max(x) &= (x_0 + \alpha) \times \beta^k \\ &= 2 \times 4^k \end{aligned}$$

So, by (6.12), we derive the invariant

$$0 \leq x \leq 2 \times 4^k$$

Defining the Strongest Conjunctive Bounding Predicate

We define a bounding predicate as a predicate which describes invariant set membership properties of a variable. For example, $x \in \mathbb{N}$ states that the variable x is bound to the set of natural numbers, and $\frac{x}{2} \in \mathbb{N}$ states that the variable x is bound to the set of even, natural numbers.

For all variables, x, accessible to an MBS loop executing under a precondition Q, if $R_{i,k+1} \Rightarrow (k \in \mathbb{N} \wedge x_{k+1} = \phi_i(x_k))$, and $x_{k+1} = \phi_i(x_k)$ under Q, yields the difference equation solution $x_i(k)$ for x, then we can say that so long as the loop iterates only through branch-i, after an arbitrary $k+1$st iteration, $k \in \mathbb{N} \wedge x = x_i(k+1)$. If the loop does not iterate at all then, by difference equation solution, $Q \Rightarrow x = x_i(0)$. Therefore, $k \in \mathbb{N} \wedge x = x_i(k)$ is invariant so long as the loop iterates only through branch-i, if it iterates at all. This implies that ${x_i}^{-1}(x) \in \mathbb{N}$ is invariant, so as long as all iterations are through branch-i.

Now, if we can show that the application of all of the update functions for x in every other branch maintains ${x_i}^{-1}(x) \in \mathbb{N}$, then ${x_i}^{-1}(x) \in \mathbb{N}$ is invariant.

For example, let $R_{i,k+1} \Rightarrow (k \in \mathbb{N} \wedge x_{k+1} = x_k + 2)$ and $Q \Rightarrow (x = 0)$, then the difference equation solution for x through branch-i only is $x(k) = 2k$. So, $k \in \mathbb{N} \wedge x = 2(k)$ is invariant so long as the loop only iterates through branch-i or not at all. Therefore, $\frac{x}{2} \in \mathbb{N}$ is invariant so long as all iterations are through branch i. Suppose that for some branch j ($j \neq i$), the update function for x is $\phi_j(x) = x + 4$. When $\frac{x}{2} \in \mathbb{N}$ holds $\frac{x+4}{2} \in \mathbb{N}$ also holds. This implies that so long as all iterations are through either branch-i or branch-j, then $\frac{x}{2} \in \mathbb{N}$ is invariant. Extending this, if the application of the update function for x, $\phi_j(x)$, from all branches $j \in [1..n]$ establishes $\frac{\phi_j(x)}{2} \in \mathbb{N}$, then $\frac{x}{2} \in \mathbb{N}$ is invariant for the loop.

Summarising, we state the following theorem:

Theorem 6.2. [Bounding Predicate Theorem]

For an MBS loop `do G` $\rightarrow$ `if` $\mathtt{C}_1 \rightarrow \mathtt{S}_1$ `[] ...[]`$\mathtt{C}_n \rightarrow \mathtt{S}_n$ `fi od`, executing under the precondition Q, for any branch i, where $R_{i,k+1} \Rightarrow (k \in \mathbb{N} \wedge x_{k+1} = \phi_i(x_k))$ and $Q \Rightarrow (x = x_i(0))$, and $x_i(k)$ is the difference equation solution for x yielded from $R_{i,k+1}$ and Q, if $\forall j \in [1..n] \left(\phi_j\left({x_i}^{-1}(x)\right) \in \mathbb{N}\right)$ when $\left({x_i}^{-1}(x)\right) \in \mathbb{N}$, then $\left({x_i}^{-1}(x)\right) \in \mathbb{N}$ is a loop invariant.

For example, consider again, the following MBS program:

```
x := 0;
do G →
    if C →
        x := x + 2;
    [] ¬C →
        x := x * 4;
    fi
od
```

From $R_{1,k+1}$, we see $\phi_1(x) = x + 2$ and from $R_{2,k+1}$, $\phi_2(x) = 4x$, and $Q \Rightarrow (x = 0)$. $R_{1,k+1}$ yields the difference equation solution, $x_1(k) = 2k$. $R_{2,k+1}$, yields the difference equation solution $x_2(k) = 0$.

From the difference equation solution from branch-1, we determine that ${x_1}^{-1}(x) \in \mathbb{N}$, that is, $\frac{x}{2} \in \mathbb{N}$, is invariant so long as all iterations are through branch 1. Now, by theorem 6.2, if we can show that $\phi_2\left({x_1}^{-1}(x)\right) \in \mathbb{N}$, when ${x_1}^{-1}(x) \in \mathbb{N}$, then ${x_1}^{-1}(x) \in \mathbb{N}$ is invariant for the loop. When $\frac{x}{2} \in \mathbb{N}$, then $4 \times \frac{x}{2} \in \mathbb{N}$, so $\frac{x}{2} \in \mathbb{N}$ is invariant.

Because $x_2(k)$ is independent of k, we do not attempt to derive an invariant from this.

From this section we have applied techniques to derive the following invariant from this loop.

$$0 \leq x \leq 2 \times 4^k \wedge \frac{x}{2} \in \mathbb{N}$$

6.4 Inference from Special Case Invariants

Consider the MBS loop:

```
{Q}
do G →
    if C_1 → S_1
    [] C_2 → S_2
    ...
    [] C_n → S_n
fi
od
```

As mentioned previously, an MBS loop has a strongly connected Branch Successor Graph. That is, every branch can be succeeded by any other branch, including itself. It is the property of cyclic succession that we wish to exploit in deriving invariants. This property implies that it possible to iterate through only one branch with every iteration of an MBS loop. For example, considering the Find First Maximum example of Program 8, if the first element of the array is the maximum, then the loop will iterate entirely through branch 1, and if the array is sorted in ascending order, then the loop iterates entirely through branch 2.

We may define a conjunctive invariant describing those properties that are invariant if a loop continually iterates only through branch i, by applying algorithm 9 (presented later in this section) to $R_{i,k+1}$. We will call such an invariant the *Branch-i Invariant*, and denote it P_i. The Branch-i Invariant is a special case of the MBS loop invariant corresponding to the instance of the branch successor graph described by Fig. 6.2. As such, the branch-i invariant, P_i, is stronger than or equivalent to the MBS loop invariant, and must therefore, imply the MBS loop invariant, denoted P.

In this section we introduce an heuristic method for inferring an invariant, P, for an MBS loop from the strong branch-i invariants.

An heuristic method for inferring a general invariant, P, for an n-branch MBS loop from the strong branch-i invariants, must be based on the property

$$\forall i \in [1..n] \, (P_i \Rightarrow P) \tag{6.19}$$

The invariant *true* is the weakest invariant P that satisfies (6.19). This is generally too weak to be of any use for verification. The strongest invariant

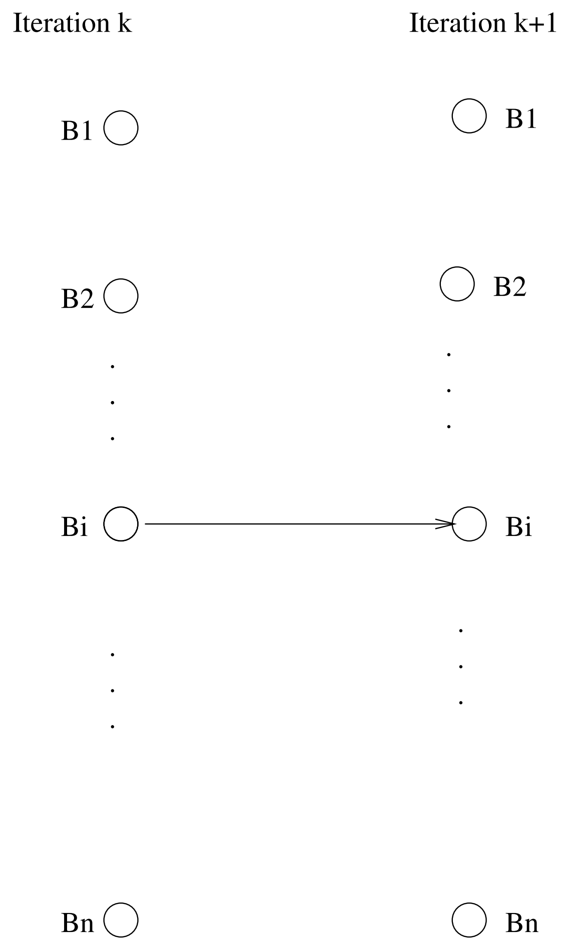


Figure 6.2: Branch-i Invariant Branch Successor Graph

that satisfies (6.19), is the disjunctive invariant $\bigvee(P_1..P_n)$. This invariant only describes the possibility that a loop may iterate through any single branch on all iterations. That is, it describes the strong invariant of a loop where every iteration must pass through branch 1 or every iteration must pass through branch 2, etc. This invariant is, generally, too strong to be of any use in reasoning about correctness as it is not an invariant for an instance of the loop iterating through an arbitrary combination of paths on successive iterations.

The problem to be solved, is one of finding a P which satisfies (6.19) and which would also be implied by every combination of branch successor, without having to perform an unbound calculation for all combinations of branch successors, and without having to introduce auxiliary variables to represent multiple iteration instances.

In this section, a number of heuristics are introduced to infer suitable invariants from only the strong branch-i invariants and the invariant facts already identified by manipulation of the iterative-form invariant[2]. These heuristics, and

[2]It is essential that these invariants be calculated first, as they provide a necessary guidance for inferring invariants from the strong branch-i invariants

techniques based on their use, provide a systematic approach to weakening the strong branch-i invariants to a conjunctive invariant, referring only to accessible variables, for any instance of the loop. The heuristic approach is based on the inference rule, (6.19).

We discuss the heuristic approach to inferring invariants from strong branch invariants with the aid of the Find First Maximum program, presented in Program 8.

6.4.1 Calculating the Branch Invariants, P_i

Calculation of the branch invariants from each $R_{i,k+1}$ can be achieved by following a similar process to that described in the previous chapter for calculating P from R_{k+1}, when there is only one path through the loop. We, however, make a change to the process defined in algorithm 7 to deal with quantified formulae. These changes are evident in algorithm 9 which describes an extension to algorithm 7 for calculating a branch invariant P_i from a branch iterative invariant $R_{i,k+1}$.

Algorithm 9 Algorithm for calculating branch invariants.

For an MBS loop of the form `do G` $\rightarrow$ `if C`$_1 \rightarrow$ `S`$_1$ `[] ...[]C`$_n \rightarrow$ `S`$_n$ `fi od`:
For every $i \in [1..n]$, given the branch-i iterative invariant, $R_{i,k+1}$, P_i is calculated as follows:

1. If $R_{i,k+1} \Rightarrow (v_{k+1} = f(w_k) \wedge F(a[f(w_k)]))$, where f is a function, F is an atomic formula, v and w are scalar variables, and a is an array variable, and if the update function for v is not consistent for all paths, then make the following substitution in $R_{i,k+1}$:

$$R_{i,k+1} \sqsubseteq (v_{k+1} = f(w_k) \wedge F(a[f(w_k)])[v/f(w_k)])$$

2. Apply algorithm 7 to $R_{i,k+1}$ to obtain P_i.

Consider the branch 1 iterative-form invariant, $R_{1,k+1}$ derived from a $k+1$st iteration through the first branch of the FFM program.

$$R_{1,k+1} \equiv k \in \mathbb{N} \wedge i_k \neq N \wedge a[p_k]_{k+1} \geq a[i_k+1]_{k+1} \wedge i_{k+1} = i_k+1 \wedge p_{k+1} = p_k \wedge a_{k+1} = a_k$$

Applying step 1 of algorithm 9, we see that p is the only index variable of a which is not consistently updated in branch 1 and 2 ($\phi_1(p) = p$ and $\phi_2(p) = p+1$). Applying the substitution, we obtain

$$\begin{aligned} R_{1,k+1} &\sqsubseteq \left(\begin{array}{l} k \in \mathbb{N} \wedge i_k \neq N \wedge \left(a[p_k]_{k+1} \geq a[i_k+1]_{k+1}\right)[p/p_k] \wedge \\ i_{k+1} = i_k + 1 \wedge p_{k+1} = p_k \wedge a_{k+1} = a_k \end{array} \right) \\ &\equiv \left(\begin{array}{l} k \in \mathbb{N} \wedge i_k \neq N \wedge \left(a[p]_{k+1} \geq a[i_k+1]_{k+1}\right) \wedge \\ i_{k+1} = i_k + 1 \wedge p_{k+1} = p_k \wedge a_{k+1} = a_k \end{array} \right) \end{aligned}$$

Applying step 2 (algorithm 7), we calculate

$$P_1 \equiv (k \in \mathbb{N} \wedge i = k+1 \wedge p = 0 \wedge k \neq N \wedge \forall j \in [0..k]\,(a[j+1] \leq a[p]))$$

This strong invariant corresponds to the special case where the loop only iterates through branch 1. This only occurs when the first element of a is the first occurrence of the maximum element in a.

By property (6.19), we know that the invariant for the loop, P must be implied by this invariant. That is,

$$(k \in \mathbb{N} \wedge i = k + 1 \wedge p = 0 \wedge k \neq N \wedge \forall j \in [0..k]\, (a\,[j + 1] \leq a[p])) \Rightarrow P$$

Similarly, we can determine that the Branch-2 Invariant, corresponding to the special case where the array is sorted in strictly ascending order, given by applying algorithm 9, is

$$P_2 \quad \equiv \quad (k \in \mathbb{N} \wedge i = k + 1 \wedge p = k + 1 \wedge k \neq N \wedge \forall j \in [0..k]\, (a\,[j] < a[p]))$$

This strong invariant must also imply the MBS loop invariant, P. Now, in order to satisfy (6.19), the invariant for the MBS loop, P must be implied by P_1 and P_2. That is,

$$\begin{array}{rcl} (k \in \mathbb{N} \wedge i = k + 1 \wedge p = 0 \wedge k \neq N \wedge \forall j \in [0..k]\, (a\,[j + 1] \leq a[p])) & \Rightarrow & P \wedge \\ (k \in \mathbb{N} \wedge i = k + 1 \wedge p = k + 1 \wedge k \neq N \wedge \forall j \in [0..k]\, (a\,[j] < a[p])) & \Rightarrow & P \end{array} \tag{6.20}$$

We begin our quest for a suitable P by identifying formulae that are implied by all branch-i invariants, and reason that any such formula must be invariant. To this end, we extend the theory of section 6.3.1. We then introduce techniques to identify invariants from relations involving sequences which can't be handled with the conditional quantification rules of section 6.3.2.

6.4.2 Common Predicates

We propose the following theorem

Theorem 6.3. [Common Predicates Theorem] For any MBS loop, yielding the branch invariants $P_1, \ldots, P_n$, if $\forall i \in [1..n]\, (P_i \Rightarrow F)$, then F is invariant.

Proof. Each P_i is a generalisation of a branch-iterative invariant $R_{i,k+1}$. If each P_i maintains F, then each corresponding $R_{i,k+1}$ must describe update functions to maintain F. So, if F holds at the beginning of any arbitrary $k + 1$st iteration ($k \geq 0$), then regardless of the path taken on the $k + 1$st iteration, F will be maintained after this iteration, and F is invariant (refer to the fundamental invariance theorem in chapter 9)[3]. Therefore, if all branch invariants, P_i, establish F, then F is invariant. □

[3]Note that we are not considering the case of the precondition implying F as this is dealt with later. In this section we are only trying to find facts that are invariant after the first iteration.

Logical and algebraic manipulations are sometimes required to identify some implied, but not explicit, common formulae, which may be useful. This is anticipated to be problematic for automating these methods.

Consider the following loop from [80]:

$\{X + Y = N\}$
```
x := X; y := Y;
do G →
    if C →
        x := x - 2;
        y := y - 2;
    [] ¬C →
        x := x - 1;
        y := y - 3;
    fi
od
```

The derived branch invariants for this loop, are

$$
\begin{aligned}
P_1 &= (k \in \mathbb{N} \wedge X + Y = N \wedge x = X - 2(k+1) \wedge y = Y - 2(k+1) \wedge C_k \wedge G_k) \\
P_2 &= (k \in \mathbb{N} \wedge X + Y = N \wedge x = X - (k+1) \wedge y = Y - 3(k+1) \wedge \neg C_k \wedge G_k)
\end{aligned}
$$

Noting that the predicate $X + Y = N$ is common to both P_1 and P_2, and that P_1 and P_2 both imply equalities between X and Y and k. We expand P_1 by manipulating $x = X - 2(k+1)$ and $y = Y - 2(k+1)$ as follows:

$$
\begin{aligned}
x = X - 2(k+1) &\equiv X = x + 2(k+1) \\
y = Y - 2(k+1) &\equiv Y = y + 2(k+1)
\end{aligned}
$$

By substitution, $(X + Y = N) \equiv (x + 2(k+1) + y + 2(k+1) = N) \equiv (x + y + 4(k+1) = N)$. We have determined that $P_1 \Rightarrow (x + y + 4(k+1) = N)$.

Similarly, $P_2 \Rightarrow (x + y + 4(k+1) = N)$. Therefore, by theorem 6.3, $x + y + 4(k+1) = N$ is invariant.

6.4.3 Deriving Universally Quantified Formula

During calculation of branch invariants, sequence properties are generalised by induction resulting in universally quantified formula of the form $\forall j \in [0..k]\,(F(a))$, where a is a sequence variable and F is a predicate.

Consider a loop, for which each branch, i, yields a branch invariant, P_i, where

$$P_i \Rightarrow \forall j \in [0..k]\,(F_i(a))$$

If $\forall j \in [0..k]\,(F_i(a))$ is not common to every P_i, then the generalisation which produced the quantified formula from the iterative-form invariant does not hold for an arbitrary combination of branch successors. That is, $\forall j \in [0..k]\,(F_i(a))$ holds if the loop iterates only through branch i, however it may not hold for any other instance of the loop, and so can't be called an invariant of the loop. A weaker formula may hold. One way to weaken a universally quantified formula is to reduce the range over which the bound variables are bound.

In this section, heuristic rules are introduced, which when applied, weaken the quantified formula implied by the branch invariants in such a way to allow a conjunctive invariant to be constructed. That is, for each quantified formula, implied by a branch invariant P_i, where the formula is of the form $\forall j \in [0..k]\,(F_i(a))$, we attempt to weaken that formula to $\forall j \in R_i\,(F_i(a))$, where $\forall i \in [1..n]\,(R_i \subseteq [0..k])$ and $\forall i, j \in [1..n]\,(i \neq j \Rightarrow R_i \cap R_j = \emptyset)$. A strong conjunctive invariant could then be composed by

$$\bigwedge_{i\in[1..n]} (\forall j \in R_i\,(F_i(a))) \tag{6.21}$$

Weakening of the quantified formulae by arbitrarily reducing the range of the bound variable is generally not sufficient for arriving at a correct loop invariant. The reduction of the quantified formulae ranges must be guided by existing, previously derived, invariants in order to arrive at a consistent and invariant set of quantified formulae describing a loop. These existing invariants may include the iterative-form invariant.

We introduce an approach for determining weakened ranges for the bound variables of a quantified formula by determining a partial order of the variables used as indices in each of the branches, and using this order to guide substitution in, and hence reduction of, the ranges. Deriving this order is introduced with the find first maximum example.

We conjecture that the techniques discussed are effective for deriving invariants for partitioning, search and sort type algorithms.

Defining a Partial Order of Index Variables

A *partial order* on a set of variables is often maintained by a program. This ordering information is generally very useful to support verification, and is essential for the verification of partitioning type problems, when the ordered variables are used to index an array. The techniques for deriving conjunctive invariants from quantified formula, presented at the end of this section, are reliant on such orderings being identified if they exist.

To introduce the technique for defining the partial order of index variables, we define the terms *updating branch* and *non-updating branch*, with respect to a variable.

Definition 10. A branch i is said to be *updating* with respect to some variable v if $v(k)$, the difference equation solution for v yielded from $R_{i,k+1}$, does not equal v_0. Conversely, a branch i is said to be *non-updating* with respect to some variable v if $v(k)$, the difference equation solution for v yielded from $R_{i,k+1}$, is equal to v_0.

Considering the find first maximum example, branches 1 and 2 are updating with respect to i, because $R_{1,k+1} \Rightarrow (i_{k+1} = i_k+1)$ and $R_{2,k+1} \Rightarrow (i_{k+1} = i_k+1)$. So from both $R_{1,k+1}$ and $R_{2,k+1}$ the difference equation solution, $i(k) = k$, is derived. Branch 1 is non-updating with respect to p, because $R_{1,k+1} \Rightarrow (p_{k+1} = p_k)$, so $p(k) = p_0$. Branch 2 is updating with respect to p, because $R_{2,k+1} \Rightarrow (p_{k+1} = i_k + 1)$, and so $p(k) = k \neq p_0$.

Consider any MBS loop, `do G` $\rightarrow$ `if C`$_1 \rightarrow$ `S`$_1$ `[] ...[]C`$_n \rightarrow$ `S`$_n$ `fi od`. Let V_a be the set of all variables used as index variables for a sequence a in the body of the loop. For all variables, $v \in V_a$, let U_v be the set of all branches i, where i is updating with respect to v. Let T be the set

$$T = \bigcup_{v \in V_a} \{(v, U_v)\} \tag{6.22}$$

Then T is the partial function from a-index variables, v, to the set of branches that update v.

For example, in the find first maximum example, i and p are all used as index variables for a in the loop. As shown above, i is updated in branches 1 and 2, so $U_i = \{1, 2\}$, and p is updated only in branch 2, so $U_p = \{2\}$. Therefore, by (6.22),

$$T = \{(i, U_i)\} \cup \{(p, U_p)\} = \{(i, \{1, 2\}), (p, \{2\})\}$$

We now define the partial order, $\subseteq$, on T. Let $O = (T, \subseteq)$ be the partial order, $\subseteq$, on the set T, where for all (i, U_i) and (j, U_j) in T, $(i, U_i) \subseteq (j, U_j) \equiv U_i \subseteq U_j$.

If for any two variables v and w, where $(v, U_v) \in T$ and $(w, U_w) \in T$, and $U_w \subseteq U_v$, then v is updated in every branch that w is updated, as well as in all of those branches in $U_v - U_w$. If the value of v after k iterations through any branch in U_w is greater than or equal to the value of w after k iterations through any branch in U_w, then if the loop does not enter any branches in $U_v - U_w$, $v \geq w$ must hold. If an additional $k + 1$st iteration occurs through any branch, i in $U_v - U_w$ and $v_i(k + 1) > v_i(k)$ (branch i increases v) for all $i \in (U_v - U_w)$, then $v > w$ must hold after $k + 1$ iterations through any branch. We can reason similarly about decreasing functions yielding $v < w$.

The poset $(T, \subseteq)$ can be represented pictorially by a Hasse diagram. The nodes represent the elements of the set T. If $a, b \in T$ and $a \subset b$ and there is no $c \in T$ such that $a \subset c \subset b$, then the node representing a is placed below the node representing b and the two nodes are joined by an edge. Constructing a Hasse diagram from O may be useful to support the above process.

The restrictions on being able to determine ordering by the above method, while appearing very strong, are often met in many types of loop. The majority of partitioning programs must maintain indices for the beginning and ends of partitions. Such partitioning is also apparent in sorting programs to represent the boundary between sorted and unsorted and searching programs to represent the boundary between searched and not searched. We conjecture that the above restrictions on being able to derive a partial ordering of variables actually characterises partitioning algorithms.

We formalise the conditions for deriving ordering information with the rules in table 6.1.

Table 6.1: Deriving Variable Order

For any two variables v and w, where $(v, U_v) \in T$ and $(w, U_w) \in T$, and $U_w \subseteq U_v$, and where $v_b(k)$ represents the difference equation solution with respect to k for v for all iterations through branch b, we evaluate the following conditions and map them to the corresponding orderings:

Condition	**Order**
$\forall b \in U_w\, (v_b(k) \geq w_b(k)) \wedge \forall b \in (U_v - U_w)\, (v_b(k+1) > v_b(k))$	$w \leq v$
$\forall b \in U_w\, (v_b(k) > w_b(k)) \wedge \forall b \in (U_v - U_w)\, (v_b(k+1) > v_b(k))$	$w < v$
$\forall b \in U_w\, (v_b(k) \leq w_b(k)) \wedge \forall b \in (U_v - U_w)\, (v_b(k+1) < v_b(k))$	$v \leq w$
$\forall b \in U_w\, (v_b(k) < w_b(k)) \wedge \forall b \in (U_v - U_w)\, (v_b(k+1) < v_b(k))$	$v < w$

Additionally we can determine the following for any variable v, where (v, U_v) is not the greatest element of $(T, \subseteq)$:

Condition	**Order**
$\forall b \in U_v\, (v_b(k+1) < v_b(k))$	$v \leq v_0$
$\forall b \in U_v\, (v_b(k+1) > v_b(k))$	$v \geq v_0$

Otherwise, ordering is non-determinable for any arbitrary loop.

Consider again the find first maximum example. We see that $O = \{((p, \{2\}), (i, \{1, 2\}))\}$, because $\{(p, \{2\})\} \subseteq \{(i, \{1, 2\})\}$. Now, $U_p = \{2\}$, and the difference equation solution for p in branch 2 is $p_2(k) = k$. The difference equation solution for i in branch 2 is, $i_2(k) = k$, and so $i_2(k) = p_2(k)$. Therefore, so long as all iterations are through branch 2, $i = p$ will hold. Now $U_i - U_p = \{1\}$ and $i_1(k) = k$, so $i_1(k+1) = k+1$, and $i_1(k+1) > i_1(k)$. We have, therefore, established the conditions matching the first rule in table 6.1

$$\forall b \in U_p\, (i_b(k) \geq p_b(k)) \wedge \forall b \in (U_i - U_p)\, (i_b(k+1) > i_b(k))$$

and derive the invariant

$$p \leq i$$

We also determine that $\forall b \in U_p\, (p_b(k+1) > p_b(k))$, that is $p_2(k+1) > p_2(k)$, and so we derive $p \geq 0$

So we can state that the following is invariant for the find first maximum loop:

$$\begin{aligned} & p \geq 0 \wedge p \leq i \\ \equiv\ & 0 \leq p \leq i \end{aligned}$$

Using the Partial Order to Guide Weakening Quantified Formulae

Considering the poset $O = (T, \subseteq)$, constructed by the technique above for the index variables for a sequence, a, the minimal elements of O, correspond to those variables that are updated by the fewest branches. In a partitioning problem, these nodes will correspond to the variables used to index the first and last partitions (if the last is constructed from the end of the sequence). We can use this information to guide range reduction in quantified formulae involving the sequence a.

For all minimal elements, (v, U_v), we reduce the range of the quantified formulae involving a in every branch $b \in U_v$ by the following. For every branch $b \in U_v$, let the difference equation solution for v yielded from $R_{b,k+1}$ be represented by $v_b(k)$, then $P_b \Rightarrow v = f(k)$, where $f(k) = v_b(k+1)$. Now, if $P_b \Rightarrow v = f(k) \wedge \forall j \in [0..k]\,(F_b(a))$, then we weaken the quantified formula by the following substitution:

$$\forall j \in \left[0..f^{-1}(v)\right](F_b(a)) \tag{6.23}$$

The resulting quantified formula is invariant, because v corresponds to a minimal element.

For example, $(p, \{2\})$ is a minimal element of the poset O for the find first maximum loop. From the calculations performed previously, we have

$$P_2 \equiv (k \in \mathbb{N} \wedge i = k+1 \wedge p = k+1 \wedge k \neq N \wedge \forall j \in [0..k]\,(a\,[j] < a[p]))$$

Weakening this quantified formula by (6.23), we see the $P_2 \Rightarrow (p = k+1)$, therefore $P_2 \Rightarrow (k = p-1)$, and by substitution we identify the invariant

$$\forall j \in [0..p-1]\,(a\,[j] < a[p])$$

For variables that are not minimal elements, under special conditions, we can derive similar invariants, by restricting the ranges of quantified formulae.

For any two variables v and w, where $U_w \subseteq U_v$ and (v, U_v) is not a minimal element of the corresponding poset O. If $v_b(k)$ represents the difference equation solution with respect to k for v, yielded from $R_{b,k+1}$, if $\forall b \in U_w\,(v_b(k) = w_b(k))$, then v is only differs from w by application of update functions in branches in $U_v - U_w$. If for each of these branches, b, $P_b \Rightarrow v = f(k) \wedge \forall j \in [0..k]\,(F_b(a))$, where f is a function, then for all elements of a between w and $f^{-1}(v)$, one of $(F_b(a))$ must hold, where $b \in (U_v - U_w)$. We derive the weakened quantified formula by the following substitution:

$$\forall j \in \left[w..f^{-1}(v)\right]\left(\bigvee_{b \in (U_v - U_w)} F_b(a)\right) \tag{6.24}$$

Considering the find first maximum, $(i, \{1, 2\})$ is the only element of the poset O that is not a minimal element. We identify the variable p as a variable establishing the condition $U_p \subseteq U_i$. Now $U_p = \{2\}$ and $U_i = \{1, 2\}$, and for all

$b \in \{2\}$, $i_b(k) = p_b(k)$. That is, $i_2(k) = k = p_2(k)$. Therefore p and i only differ when i is updated in branch 1. From the calculations performed previously, we have

$$P_1 \equiv (k \in \mathbb{N} \wedge i = k+1 \wedge p = 0 \wedge k \neq N \wedge \forall j \in [0..k]\,(a\,[j+1] \leq a[p]))$$

From this we establish that $k = i - 1$, and applying the substitution in (6.24), we obtain the invariant

$$\forall j \in [p..i-1]\,(a\,[j+1] \leq a[p])$$

Now, from the techniques so far presented in this section we have been able to identify the following invariants:

$$\begin{array}{l} k \in \mathbb{N} \wedge i = k+1 \wedge k \neq N \\ 0 \leq p \leq i \\ \forall j \in [0..p-1]\,(a\,[j] < a[p]) \\ \forall j \in [p..i-1]\,(a\,[j+1] \leq a[p]) \end{array}$$

As each of these are invariant, their conjunction must also be invariant. We have, therefore, derived the following invariant which we will denote P:

$$\begin{array}{rcl} P & \equiv & \left(\begin{array}{l} k \in \mathbb{N} \wedge i = k+1 \wedge k \neq N \wedge 0 \leq p \leq i \wedge \\ \forall j \in [0..p-1]\,(a\,[j] < a[p]) \wedge \forall j \in [p..i-1]\,(a\,[j+1] \leq a[p]) \end{array}\right) \\ & \equiv & \left(\begin{array}{l} i \in \mathbb{N}^+ \wedge i-1 \neq N \wedge 0 \leq p \leq i \wedge \\ \forall j \in [0..p-1]\,(a\,[j] < a[p]) \wedge \forall j \in [p..i-1]\,(a\,[j+1] \leq a[p]) \end{array}\right) \end{array}$$

We can apply (5.26) to this to derive the conjunctive postcondition

$$R \equiv \left(\begin{array}{l} i \in \mathbb{N}^+ \wedge i = N \wedge 0 \leq p \leq i \wedge \\ \forall j \in [0..p-1]\,(a\,[j] < a[p]) \wedge \forall j \in [p..i-1]\,(a\,[j+1] \leq a[p]) \end{array}\right)$$

As illustrated in chapter 9, these conjunctive invariants and postconditions are strong enough to support the direct method of verification.

Invariant Consistency In the above process, the derived invariants are conjoined to construct the conjunctive invariant. In conjoining these invariants, any contradiction indicates loop inconsistency.

We summarise this section by presenting algorithm 10 for deriving quantified formulae invariants for partitioning problems.

6.5 Conclusion

In concluding this chapter, we present an algorithm which combines all of the techniques discussed in this chapter for constructing a strong, conjunctive invariant and postcondition from the code of an MBS loop. This algorithm is used to derive a conjunctive invariant and postcondition from the Dutch National Flag partition [26].

Algorithm 10 Deriving Partition Related Invariants

For an MBS loop `do G` $\rightarrow$ `if` $\mathtt{C}_1 \rightarrow \mathtt{S}_1$ `[] ...[]` $\mathtt{C}_n \rightarrow \mathtt{S}_n$ `fi od`,

1. Calculate R_{k+1}, $R_{i,k+1}$ and P_i by the method of algorithm 7.
2. For every sequence variable a, where $\forall i \in [1..n] \, (P_i \Rightarrow \forall j \in [0..k] \, (F_i(a)))$, where $F_1(a), \ldots, F_n(a)$ are predicates involving a.
   (a) Calculate by (6.22), the set T, for all index variables of a.
   (b) For all minimal elements of T, (v, U_v), for every branch $i \in U_v$ extract the quantified formula $\forall j \in [0..k] \, (F_i(a))$ from P_i and state the following equivalent formula as an invariant
   $$\forall j \in \left[0..f^{-1}(v)\right] (F_i(a))$$
   where $P_i \Rightarrow v = f(k)$.
   (c) For elements of T, (v, U_v), that are not minimal elements, if $\exists \, (w, U_w) \in T \, (U_w \subseteq U_v)$ and $\forall b \in U_w \, (v_b(k) = w_b(k))$, then for every branch $i \in U_v$ extract the quantified formula $\forall j \in [0..k] \, (F_i(a))$ from P_i and state the following equivalent formula as an invariant
   $$\forall j \in \left[w..f^{-1}(v)\right] \left(\bigvee_{b \in (U_v - U_w)} F_i(a) \right)$$
   where $P_i \Rightarrow v = f(k)$.
3. Conjoin all quantified formula from step 2 to the invariant P. If there is a contradiction, then the loop is inconsistent in the maintenance of an invariant.

6.5.1 Dutch National Flag Example[4]

Consider the Dutch National Flag partition (Program 9) from [26].

We apply algorithm 11 to derive a conjunctive invariant and postcondition from the loop, and hence the program.

Step 1:

We begin with $P = true$.

Step 2:

The loop precondition is calculated by applying the *sp* predicate transformer as

$$(r = 0 \land w = 0 \land b = N + 1 \land a \in \texttt{ seq } COLOUR \land \#a = N \land N > 0)$$

[4] This is a program taken from R.G. Dromey, *Program Derivation: The Development of Programs from Specifications*, Addison-Wesley, pp438-441, 1989. In this context array variables are considered to be indexed starting from 1 not 0.

Algorithm 11 Deriving the strongest accessible conjunctive invariant and postcondition.

For an MBS loop `do G` $\rightarrow$ `if` $\texttt{C}_1 \rightarrow \texttt{S}_1$ `[] ...[]`$\texttt{C}_n \rightarrow \texttt{S}_n$ `fi od`,

1. Begin with $P \equiv true$.
2. Calculate R_{k+1}, and each $R_{i,k+1}$.
3. Solve all solvable difference equations for each $R_{i,k+1}$.
4. If R_{k+1} implies that there are invariant conditional quantified formulae, then conjoin these formula to P. (See section 6.3.2.)
5. By the method of section 6.3.3, identify and conjoin to P any invariants describing the range of values of a variable.
6. Calculate each P_i by the method of Algorithm 9.
7. For any quantifier-free formula F, where $\forall i \in [1..n]\,(P_i \implies F)$, conjoin that formula to P.
8. Calculate any partial orderings between variables used as indices for arrays. (See section 6.4.3.)
9. Apply Algorithm 10 to identify and conjoin to P any quantified formula yielded from partitioning problems.
10. State P as the strongest accessible conjunctive loop invariant.
11. Calculate R, the strongest accessible conjunctive postcondition, by (5.26)

The iterative form invariant for the loop is calculated as:

$$R_{k+1} \equiv k \in \mathbb{N} \wedge \left(\begin{array}{l} \left(\begin{array}{l} r_{k+1} = r_k \wedge w_{k+1} = w_k + 1 \wedge b_{k+1} = b_k \wedge w_k \neq b_k - 1 \wedge \\ a\left[w_k+1\right]_k = WHITE \wedge \forall j \in [0..\#a-1]\,(a[j]_{k+1} = a[j]_k) \end{array} \right) \vee \\ \left(\begin{array}{l} r_{k+1} = r_k \wedge w_{k+1} = w_k \wedge b_{k+1} = b_k - 1 \wedge w_k \neq b_k - 1 \wedge \\ a\left[w_k+1\right]_k = BLUE \wedge a\left[b_k-1\right]_{k+1} = a\left[w_k+1\right]_k \wedge a\left[w_k+1\right]_{k+1} = a\left[b_k-1\right]_k \end{array} \right) \vee \\ \left(\begin{array}{l} r_{k+1} = r_k + 1 \wedge w_{k+1} = w_k + 1 \wedge b_{k+1} = b_k \wedge w_k \neq b_k - 1 \wedge \\ a\left[w_k+1\right]_k = RED \wedge a\left[r_k+1\right]_{k+1} = a\left[w_k+1\right]_k \wedge a\left[w_k+1\right]_{k+1} = a\left[r_k+1\right]_k \end{array} \right) \end{array} \right)$$

Therefore we identify three branch-iterative invariants (simplified to remove as many references as possible to kth iteration array instances):

$$\begin{array}{lcl} R_{1,k+1} & = & \left(\begin{array}{l} k \in \mathbb{N} \wedge r_{k+1} = r_k \wedge w_{k+1} = w_k + 1 \wedge b_{k+1} = b_k \wedge w_k \neq b_k - 1 \wedge \\ a\left[w_k+1\right]_{k+1} = WHITE \end{array} \right) \\ R_{2,k+1} & \equiv & \left(\begin{array}{l} k \in \mathbb{N} \wedge r_{k+1} = r_k \wedge w_{k+1} = w_k \wedge b_{k+1} = b_k - 1 \wedge w_k \neq b_k - 1 \wedge \\ a\left[b_k-1\right]_{k+1} = BLUE \end{array} \right) \\ R_{3,k+1} & \equiv & \left(\begin{array}{l} k \in \mathbb{N} \wedge r_{k+1} = r_k + 1 \wedge w_{k+1} = w_k + 1 \wedge b_{k+1} = b_k \wedge w_k \neq b_k - 1 \wedge \\ a\left[r_k+1\right]_{k+1} = RED \end{array} \right) \end{array}$$

Step 3:

Program 9 Dutch National Flag Partition (DNF)

$Q \equiv (a \in seq\ \texttt{COLOUR} \wedge \#a = N \wedge N > 0)$

```
dnf(a : INOUT seq COLOUR) {

    r:=0; w:=0; b:=N+1;
    do w ≠ b-1 →
        if a[w+1] = WHITE →
            w := w + 1;
        [] a[w+1] = BLUE →
            swap(a[w+1],a[b-1]);
            b := b-1;
        [] a[w+1] = RED →
            swap(a[w+1],a[r+1]);
            w := w + 1;
            r := r + 1;
        fi;
    od;
}
```

For each $R_{i,k+1}$ we compute the following difference equation solutions.

From $R_{1,k+1}$:

$$\begin{array}{lcr} w_{k+1} = w_k + 1 & \Longrightarrow & w\,(k) = k \\ r_{k+1} = r_k & \Longrightarrow & r\,(k) = r_0 = 0 \\ b_{k+1} = b_k & \Longrightarrow & b\,(k) = b_0 = N + 1 \end{array}$$

From $R_{2,k+1}$:

$$\begin{array}{lcr} w_{k+1} = w_k & \Longrightarrow & w(k) = w_0 = 0 \\ r_{k+1} = r_k & \Longrightarrow & r(k) = r_0 = 0 \\ b_{k+1} = b_k - 1 & \Longrightarrow & b(k) = N + 1 - k \end{array}$$

From $R_{3,k+1}$:

$$\begin{array}{lcr} w_{k+1} = w_k + 1 & \Longrightarrow & w(k) = k \\ r_{k+1} = r_k + 1 & \Longrightarrow & r(k) = k \\ b_{k+1} = b_k & \Longrightarrow & b(k) = b_0 = N + 1 \end{array}$$

Step 4:

If R_{k+1} implies that there are invariant conditional quantified formulae, then conjoin these formula to P. By the method described in section 6.3.2, we see that there are no conditional formula implied by R_{k+1}, because all of the predicates describing a are dependent on conditions involving a. Therefore, P does not change.

Step 5:

Through application of (6.13) we determine the following ranges for r and w:

$$\begin{array}{l} 0 \leq w \leq k + 1 \\ 0 \leq r \leq k + 1 \end{array}$$

Applying (6.15) we calculate the following range for b:

$$N - k \leq b \leq N + 1$$

These range invariants are conjoined to P, giving

$$P \equiv (0 \leq w \leq k + 1 \wedge 0 \leq r \leq k + 1 \wedge N - k \leq b \leq N + 1)$$

Step 6:

We apply algorithm 9 to calculate P_1, P_2 and P_3.

$$\begin{aligned}
P_1 &\equiv (k \in \mathbb{N} \wedge r = 0 \wedge w = k + 1 \wedge b = N + 1 \wedge \forall j \in [0..k]\,(a[j + 1] = WHITE) \wedge w \neq b) \\
P_2 &= (k \in \mathbb{N} \wedge r = 0 \wedge w = 0 \wedge b = N - k \wedge \forall j \in [0..k]\,(a[N - j] = BLUE) \wedge w \neq b + 1) \\
P_3 &\equiv (r = k + 1 \wedge w = k + 1 \wedge b = N + 1 \wedge \forall j \in [0..k]\,(a[j + 1] = RED) \wedge w \neq b)
\end{aligned}$$

Step 7:

The only formulae implied by P_1, P_2 and P_3 are $k \in \mathbb{N}$ and $w \neq b$. Conjoining these invariants to P, we get

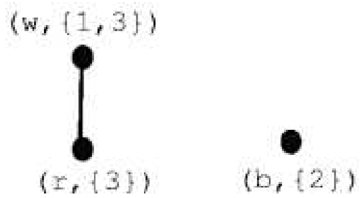
$$P \equiv (k \in \mathbb{N} \wedge 0 \leq w \leq k + 1 \wedge 0 \leq r \leq k + 1 \wedge N - k \leq b \leq N + 1 \wedge w \neq b)$$

Step 8:

We now determine the partial order of the variables used to index the array a, by applying the technique described in section 6.4.3.

We see that w, r and b are all used as index variables for a in the loop. Branches 1 and 3 are updating with respect to w, branch 3 is updating with respect to r, and branch 2 is updating with respect to b. So, $U_w = \{1, 3\}$, $U_r = \{3\}$, $U_b = \{2\}$, and $T = \{(w, \{1, 3\}), (r, \{3\}), (b, \{2\})\}$.

The following Hasse diagram depicts the poset $(T, \subseteq)$:



There is an edge from $(w, \{1, 3\})$ to $(r, \{3\})$, so $\{(r, \{3\})\} \subseteq \{(w, \{1, 3\})\}$. In branches 1 and 3 the difference equation solutions for w are $w_1(k) = k$ and $w_3(k) = k$, respectively, and the difference equation solution for r on branch 3 is $r_3(k) = k$. Evaluating the conditions from table 6.1, we see that $U_r = \{3\}$ and $w_3(k) = k = r_3(k)$, and $U_w - U_r = \{1\}$ and $w_1(k+1) = k+1$ so $w_1(k+1) > w_1(k)$. Therefore, we match the condition

$$\forall b \in U_r\,(w_b(k) \geq r_b(k)) \wedge \forall b \in (U_w - U_r)\,(w_b(k + 1) > w_b(k))$$

and derive the invariant

$$r \leq w$$

We also determine that $\forall b \in U_r\,(r_b(k+1) > r_b(k))$, and so we derive $r \geq 0$ as $(r_0 = 0)$. Similarly we determine that $w \geq 0$ and $b \leq N+1$.

So we can state that the following invariant:

$$\begin{array}{ll} & r \geq 0 \wedge w \geq 0 \wedge b \leq N+1 \wedge r \leq w \\ \equiv & 0 \leq r \leq w \wedge b \leq N+1 \end{array}$$

Conjoining this invariant to P gives

$$P \equiv \left(\begin{array}{l} k \in \mathbb{N} \wedge 0 \leq w \leq k+1 \wedge 0 \leq r \leq k+1 \wedge N-k \leq b \leq N+1 \wedge w \neq b \wedge \\ 0 \leq r \leq w \end{array} \right)$$

Step 9:

We now apply algorithm 10 to identify and conjoin to P any quantified formula exhibiting properties describing a partition.

We identify $(r, \{3\})$ and $(b, \{2\})$ as the minimal element of the poset $(T, \subseteq)$ describing the index variable ordering for a. Now, r is only updated by branch 3. We see that

$$P_3 \Rightarrow r = k+1 \wedge \forall j \in [0..k]\,(a[j+1] = RED)$$

From $r = k+1$, we derive $k = r-1$, and substitute in the quantified formula, as defined in (6.23), giving the invariant

$$\begin{array}{ll} & \forall j \in [0..r-1]\,(a[j+1] = RED) \\ \equiv & \forall j \in [1..r]\,(a[j] = RED) \end{array}$$

Similarly, we substitute $N-b$ for k in the quantified formula from branch 2, giving the invariant

$$\begin{array}{ll} & \forall j \in [0..N-b]\,(a[N-j] = BLUE) \\ \equiv & \forall j \in [b..N]\,(a[j] = BLUE) \end{array}$$

$(w, \{1,3\})$ is not a minimal element of the corresponding partial order. $U_w - U_r = 1$, and $P_1 \Rightarrow w = k+1 \wedge \forall j \in [0..k]\,(a[j+1] = WHITE)$. We know that $w_3(k) = r_3(k)$, so we can apply the substitution in (6.24) to obtain the invariant

$$\begin{array}{ll} & \forall j \in [r..w-1]\,(a[j+1] = WHITE) \\ \equiv & \forall j \in [r+1..w]\,(a[j] = WHITE) \end{array}$$

Conjoining these invariants to P, we obtain

$$P \equiv \left(\begin{array}{l} k \in \mathbb{N} \wedge 0 \leq w \leq k+1 \wedge 0 \leq r \leq k+1 \wedge N-k \leq b \leq N+1 \wedge w \neq b \wedge \\ 0 \leq r \leq w \wedge \forall j \in [1..r]\,(a[j] = RED) \wedge \forall j \in [r+1..w]\,(a[j] = WHITE) \wedge \\ \forall j \in [b..N]\,(a[j] = BLUE) \end{array} \right)$$

Step 10: The strongest accessible conjunctive loop invariant, is therefore,

$$P \equiv \left(\begin{array}{l} k \in \mathbb{N} \wedge 0 \leq w \leq k+1 \wedge 0 \leq r \leq k+1 \wedge N-k \leq b \leq N+1 \wedge w \neq b \wedge \\ 0 \leq r \leq w \wedge \forall j \in [1..r]\,(a[j] = RED) \wedge \forall j \in [r+1..w]\,(a[j] = WHITE) \wedge \\ \forall j \in [b..N]\,(a[j] = BLUE) \end{array} \right)$$

Step 11:

The strongest accessible conjunctive postcondition, R, is given by

$$\begin{array}{rcl} R & \equiv & (\neg G \wedge (Q \vee P)) \\ & \equiv & (w = b-1 \wedge (r = 0 \wedge w = 0 \wedge b = N+1 \vee P)) \\ & \equiv & \left(\begin{array}{l} k \in \mathbb{N} \wedge 0 \leq w \leq k+1 \wedge 0 \leq r \leq k+1 \wedge N-k \leq b \leq N+1 \wedge w = b-1 \wedge \\ 0 \leq r \leq w \wedge \forall j \in [1..r]\,(a[j] = RED) \wedge \forall j \in [r+1..w]\,(a[j] = WHITE) \wedge \\ \forall j \in [b..N]\,(a[j] = BLUE) \end{array} \right) \end{array}$$

This conjunctive postcondition is useful for direct proof with respect to the specification given in [26][5], by reasoning that

$$R \Rightarrow (a[1..r] = RED \wedge a[r+1..b-1] = WHITE \wedge a[b..N] = BLUE)$$

In this chapter the ideas developed in the previous chapter have been extended to deal with loops that have branching loop bodies. We explain in detail that the inductive approach of the previous chapter is not applicable to derive invariants from such loops. A normal form for loops known as multiply-branched statement form [28, 80] is introduced. We assume this normal form in order to discuss an approach that uses heuristics for deriving invariants from loops containing branches. We introduce heuristics which can be used to derive invariants for loops describing conditional functions, partitions and searches and for deriving predicates describing invariant variable ranges. Some of these heuristics can be applied directly to the branch-iterative invariants while others require the calculation of special-case branch invariants. We present Algorithm 9 for calculating these branch invariants. This algorithm is currently implemented in the prototype tool. Algorithms 10 and 11 describe the use of the heuristics and the orders in which they need to be applied to generate invariants from MBS loops. Algorithm 10 relies on complex pattern matching and simplification procedures to mechanically determine which heuristic rules to apply. This pattern matching and simplification is not currently implemented but is being worked on. The prototype supports the final algorithm (Algorithm 11) only to the extent that the calculation of the iterative-form invariant is automated. That is, at present, steps 1 and 2 are automated and the remainder are manually applied, as illustrated in the concluding example.

[5]Note the specification in [26] also states that a after the partition is a permutation of what it was previously. At this stage, such information is not derivable by the method presented. This part of the postcondition can, however, be proven by induction using the iterative-form invariant (see chapter 9).

Chapter 7

Deriving Termination Conditions

7.1 Introduction

Program verification involves proving a program satisfies its specification if it terminates, and proving that it terminates.

Chapters 4, 5 and 6 dealt with the derivation of information to support reasoning about partial correctness. This chapter deals with derivation of semantic information to support reasoning about termination. Several theorems and methods are proposed that support reasoning about loop progress and termination.

Techniques for proof of termination of imperative programs [34, 72, 2, 4, 24][1] are well understood. Two conditions must be met for a loop to terminate. (i) The loop must make progress toward establishing the termination condition described by the loop guard, and (ii) the loop guard must be strong enough to force termination after a finite number of iterations.

Existing formal techniques for investigating loop progress and termination use either variant function conditions [34, 72, 2], or weakest precondition calculation [4, 24]. Weakest precondition techniques are primarily used for program derivation. For this reason the majority of comparative discussion in this chapter concerns only variant function techniques.

For variant function techniques, proof of termination involves identifying a *variant function* (discussed in the next section) and reasoning that it is decreasing with each iteration of the loop body. A method to use a variant function and the iterative invariant (see Chap. 3 and 4) to compute loop progress and termination conditions is presented and discussed.

Dromey and Billington [27], provide an algorithm for calculating a variant function from the loop guard and precondition alone using inference rules. In section

[1] This list is far from exhaustive

7.2.1 this technique is utilised and its limitations are discussed. In an attempt to overcome the identified limitations, the existence of a general variant function is proved. The use of the general variant function, for this method, is shown to be impractical for reasoning about progress and termination, due to the complexity of the calculations required.

To overcome the problems associated with variant function methods of yielding termination information, a technique is proposed for investigating loop progress in the absence of any variant function. It can always be applied and yields the conditions under which a loop will fail to progress directly from the code. This technique comprises heuristic algorithms that can be automated as well as applied manually. It is discussed in section 7.3.

In section 7.4 we present a further technique, limited in applicability, but which yields all terminations and progress information directly from the code.

7.2 Investigating Progress with Variant Functions

Variant Function techniques are based on *well-founded sets*. A well-founded set is a partially ordered set with no infinite decreasing sequences [72]. In a proof of termination, the verifier is required to identify a function, t, which maps program variables onto the well-founded set. The set of natural numbers with the ordering, $<$, is often chosen as this well-founded set. A variant function t, is suitable when t evaluated at the start of any iteration maps to a natural number that is greater than t evaluated at the end of the iteration, for every iteration in which progress is made toward termination. In reasoning about loop progress, the verifier must, therefore, demonstrate that the variant function decreases under the statements of the loop body, with every iteration. By demonstrating that the range of t is a well-founded set, the verifier ensures that the variant function must always have a lower bound and the loop must, at some stage, terminate.

Loops fail to terminate when the statements in the loop body fail to decrease the variant function, or when the loop initialisation, loop guard and loop body imply that the variant is not bound to a well-founded set. In summary, loop progress and termination can be investigated by exploiting the formal properties of a variant function.

We define a variant function as being a function which maps loop guard variables to the set of natural numbers. We denote the variant function by t, where $t : \mathbb{P}\ V(G) \rightarrow \mathbb{N}$. In accordance with the notation presented in chapter 2, t_k represents the function t evaluated at the end of the kth iteration.

Expanding on the termination rules of [34, 72, 2], we define rules for progress based on an identifiable variant function, t as follows:

$$t_k, t_{k+1} \in W \wedge t_{k+1} < t_k, \text{ where } W \text{ is a well-founded set } (\mathbb{N}, <), \text{ and } k \in \mathbb{N}. \tag{7.1}$$

The variant conditions for loop progress can be stated more simply as

$$0 \leq t_{k+1} < t_k, \text{ where } k \in \mathbb{N}. \tag{7.2}$$

This states that the variant function, t, evaluates to a natural number greater than zero at the start of every iteration of a loop and decreases with every iteration, but never below 0.

The conditions defined by (7.2) rely on the comparison of variables on successive loop body iterations. Such information is described by the iterative-form invariant R_{k+1}. The iterative form invariant was discussed in chapter 4, and algorithms for its calculation described. The iterative-form invariant defines equalities between $k+1$st iteration instances of a variable and kth iteration instances of that variable, for every program variable accessible to the loop. Using this information (7.2) can be assessed with respect to the iterative-form invariant for a loop to determine the initial conditions required for loop progress on any iteration. That is, to identify an invariant required for loop progress.

Formally, for any loop, $\{Q\}$ `do` $G \rightarrow S$ `od`, with an iterative-form invariant, $R_{k+1} = sp\,(Q_{k+1}, S)$, and a suitable variant function, t, we can calculate the conditions required for loop progress, PC, by evaluating

$$PC \equiv 0 \leq t_{k+1} < t_k \wedge R_{k+1} \tag{7.3}$$

This produces a disjunctive predicate with references to both $k+1$st and kth iteration variable instances. As we wish to discover only the conditions required for progress at the beginning of any iteration, we attempt to substitute out all references to the $k+1$st iteration variable instances. This is simple due to the nature of the iterative-form invariant, R_{k+1}. For every variable, x, in the loop guard, loop body and loop precondition, there will be a corresponding equality in each $R_{i,k+1}$ of the form $x_{k+1} = E$, where E is an expression. These equalities are therefore maintained in each of the disjuncts of PC. We can remove all references to $k+1$st iteration variable instances from the progress conditions, PC, by the substitution

$$PC_i\left[E/x_{k+1}\right], \text{for all } x \in V(Q) \cup V(G) \cup V(S). \tag{7.4}$$

where $PC_i \implies x_{k+1} = E$, for each disjunct, PC_i, of PC.

This substitution is always possible as R_{k+1} implies equalities of the form $v_{k+1} = E$ for all variables v, where E can be reduced by recursive substitution to a function of kth iteration variable instances and constants. The iteration count k has been selected arbitrarily, and so PC after (7.4) defines the progress conditions at commencement of any $k+1$st iteration (by the law of *universal generalisation*). As there are no references to any variable instances other than kth iteration instances and constants, we can make PC more readable by dropping the variable instance subscripts and state that PC must hold at commencement of any iteration for progress to occur.

As an example, consider a loop with a variant function, $t(i) = 100-i$, and given that $R_{k+1} \implies i_{k+1} = i_k + 1$, the following progress conditions are derived by applying (7.3):

$$PC \equiv (0 \leq 100 - i_{k+1} < 100 - i_k \wedge i_{k+1} = i_k + 1)$$

Applying (7.4) to this, we obtain

$$\begin{aligned} PC &\equiv (0 \leq 100 - i_{k+1} < 100 - i_k \wedge i_{k+1} = i_k + 1)\,[i_k + 1/i_{k+1}] \\ &\equiv (0 \leq 100 - i_k - 1 < 100 - i_k \wedge i_k + 1 = i_k + 1) \\ &\equiv (i_k \leq 99) \end{aligned}$$

Therefore, $i_k \leq 99$ describes the conditions required for the associated loop to progress toward termination on any arbitrary $k+1$st iteration, ($k \in \mathbb{N}$). That is, i at the start of any $k+1$st iteration of the loop body must be less than or equal to 99 for the loop to progress toward termination. So, if $i_k > 99$, progress will not be achieved. Under these conditions, either the loop guard is negated $i_k = 100$ and termination is forced or $i_k > 100$ ensuring termination will never occur.

Because PC has been derived for an arbitrary $k+1$st iteration of the loop body, we allow ourselves to drop the k subscripts and describe the resulting progress conditions PC as an invariant that must hold on every iteration.

We are now in a position to formalise the property, that for a loop to terminate, before any iteration the program variables must either be in a state that enables progress toward termination, or in a state that forces termination. PC describes all of the states required for progress toward termination, and $\neg G$ describes the states required to force termination, so an invariant for progress or termination, TC, is:

$$TC = PC \vee \neg G \tag{7.5}$$

To summarise, we have the following theorem.

Theorem 7.1. [Variant Conditions Method]

For any loop, $\{Q\}$ `do` $G \rightarrow S$ `od`, apply the following steps:

1. Calculate the iterative-form invariant.

2. Identify a suitable variant function, t.

3. Calculate the progress conditions, PC, by (7.3).

4. Simplify the progress conditions, by applying (7.4) to substitute equal expressions for all $k+1$st iteration variable instances. Drop all k subscripts from the simplified progress conditions.

5. Calculate the failure to progress conditions, FP by negating PC.

Then, PC describes the conditions required for loop progress on any iteration, FP describes the conditions under which the loop will fail to progress toward termination. When the conjunction $Q \wedge FP$ is satisfiable then there are initial conditions allowable under the given precondition that will result in S failing to progress toward termination. The states satisfying $Q \wedge FP$ describe these initial conditions. Conversely, the progress conditions PC, conjoined to the precondition Q, describes the required initial conditions in order for S to progress toward achieving termination with every iteration. $Q \wedge PC$ is, therefore, the weakest precondition, and PC is the weakest invariant required for loop progress on every iteration. Conditions for progress or termination, TC, are defined by $TC = PC \vee \neg G$.

Corollary 7.1. If $PC \equiv \neg G$ ($FP \equiv G$), then the loop will always progress toward termination, whenever the guard G holds. If $PC \equiv G$ ($FP \equiv \neg G$), then the loop will never progress toward termination.

Corollary 7.2. Any failure to terminate defects can be corrected in one of the following ways: (1) By strengthening the precondition from Q to $Q \wedge (\neg G \vee PC)$ (see (7.5)), if acceptable, (2) By modifying S so that progress is possible under the precondition Q, or (3) By strengthening the guard to $G \wedge PC$.

Consider the Greatest Common Divisor example in Program 10.

Program 10 GCD Example

```
procedure gcd(x,y:  INOUT ℤ) {
    do x ≠ y →
       if x > y →
          x := x-y;
       [] y > x →
          y := y-x;
       fi;
    od
}
```

This example is presented by Manna in [72] as a non-trivial example to illustrate termination proof using well-founded sets. Manna assumes the precondition $X > 0 \wedge Y > 0$ for this algorithm, where X and Y represent the actual parameter values of x and y respectively. Manna's proof of termination [72] yields a statement declaring that termination is achieved under the given precondition. Were the precondition weaker, and termination not achieved, the method of proof described provides no constructive information useful for correcting the defect that caused termination failure, whether that be a defect in the program or specification. This is indicative of all existing variant function methods of investigating termination. They return a result of either $true$ or $false$.

We assume the weakest possible precondition *true* for the GCD algorithm, and demonstrate that applying the *Variant Conditions Method* yields a constructive description of conditions required for loop progress, and that these conditions can be used to reason about termination, and as constructive guidance for correcting defects in the program and specification.

Step 1: Calculate the iterative form invariant

Applying (4.1) then (4.2) to the loop code yields the iterative form invariant

$$R_{k+1} \equiv k \in \mathbb{N} \wedge \left(\begin{array}{l} x_{k+1} = x_k - y_k \wedge y_{k+1} = y_k \wedge x_k > y_k \vee \\ x_{k+1} = x_k \wedge y_{k+1} = y_k - x_k \wedge y_k > x_k \end{array} \right)$$

Step 2: Identify a suitable variant function, t

We have identified the variant function t(x,y)=x+y.

Step 3: Calculate the progress conditions, PC

$$\begin{array}{rcl} PC & \equiv & 0 \leq t_{k+1} < t_k \wedge R_{k+1} \\ & \equiv & \left[\begin{array}{l} k \in \mathbb{N} \wedge 0 \leq x_{k+1} + y_{k+1} < x_k + y_k \wedge \\ \left(\begin{array}{l} x_{k+1} = x_k - y_k \wedge y_{k+1} = y_k \wedge x_k > y_k \vee \\ x_{k+1} = x_k \wedge y_{k+1} = y_k - x_k \wedge y_k > x_k \end{array} \right) \end{array} \right] \end{array}$$

Step 4: Simplify the progress conditions, by applying (7.4)

$$\begin{array}{rcl} PC & \equiv & k \in \mathbb{N} \wedge \left(\begin{array}{l} y_k < x_k \wedge 0 \leq x_k < x_k + y_k \vee \\ x_k < y_k \wedge 0 \leq y_k < x_k + y_k \end{array} \right) \\ & \equiv & k \in \mathbb{N} \wedge (0 < y_k < x_k \vee 0 < x_k < y_k) \\ & \equiv & k \in \mathbb{N} \wedge (x_k \neq y_k \wedge 0 < x_k \wedge 0 < y_k) \end{array}$$

As we have calculated these conditions for any arbitrary iteration of the loop body, they are independent of k, the iteration count. We, therefore, drop the k subscripts to give a more natural representation of the conditions.

$$PC \equiv x \neq y \wedge 0 < x \wedge 0 < y$$

So, whenever the guard is not negated ($x \neq y$) and x and y are both greater than zero at the start of any iteration then the loop will make progress toward termination on that iteration.

Step 5: Calculate the failure to progress conditions, FP.

The failure to progress conditions are the negation of the progress conditions. So,

$$\begin{array}{rcl} FP & \equiv & \neg PC \\ & \equiv & x = y \vee x \leq 0 \vee y \leq 0 \end{array}$$

Whenever either the guard is negated ($x = y$), or either of x or y are less than or equal to zero at the start of any iteration then the loop will fail to make progress toward termination on that iteration. The negation of the guard is not of concern as the loop is expected to not progress on termination, however, the other two states must be considered.

In order for progress to be made on an iteration, the progress conditions state that $x \neq y \wedge x > 0 \wedge y > 0$ must hold. So, $x \neq y \wedge x > 0 \wedge y > 0$ is an invariant required for loop progress, and by (7.5) $x = y \vee x > 0 \wedge y > 0$ is an invariant for progress or termination. It can be reasoned that if $x > 0 \wedge y > 0$ is maintained initially then it will continue to be maintained under any iteration of the loop body and on termination.

The loop initialisation under the weakest precondition *true* yields the loop precondition $(x = X \wedge y = Y \wedge x, y \in \mathbb{Z})$. Reasoning about correctness, we note that this precondition does not exclude the states described by FP. Therefore, the loop may be entered with initial values for x and y that will cause failure to progress.

Corollary 7.2 says that we can use the derived progress conditions to correct the defect causing non-termination by strengthening the precondition. In this situation the defect is not in the program but in the program's specification. By strengthening the precondition to

$$\begin{aligned} & (x = X \wedge y = Y \wedge x, y \in \mathbb{Z} \wedge (x = y \vee x > 0 \wedge y > 0)) \\ \equiv \quad & \big(x = X \wedge y = Y \wedge \big(x = y \vee x, y \in \mathbb{N}^{+}\big)\big) \end{aligned}$$

we have reasoned that the loop will always progress toward termination. ■

In practice, a variant function is often not available when verifying code. This is particularly so for legacy code. In order to automate the method above, an investigation into techniques of deriving correct variant functions from code syntax alone was conducted.

7.2.1 Methods to Obtain Variant Functions

Dromey and Billington [27] introduce a process for determining a variant function from a loop's precondition and guard. This process can be mechanised. In some cases, however, there is not sufficient information provided by the loop precondition and loop guard to apply the process. These cases are usually characterised by loop guards with $\neq$ as the main connective where at least one of the variables or constants in the loop guard is not initialised within the scope of the program being analysed. For such variables, the initialised values or ranges are unknown. The Greatest Common Divisor program of the previous section is an example of this. As we are provided with no documentation for this function, we must make the precondition the weakest possible precondition, *true*. The initial values of x and y are dependent on the parameters passed to them. If we let X and Y represent the actual parameter values passed to x and y respectively, then the loop precondition is $(x = X \wedge y = Y \wedge x, y \in \mathbb{Z})$. This is too weak to be of any use with the variant calculation process in [27].

Attempts were made by this author to modify the process of variant calculation, however, the generation of a variant function under weak initial conditions was not achieved. This led to an attempt to describe a general variant function, applicable to all loops.

7.2.2 Toward A General Variant Function

For any loop, $\{Q\}$ `do` $G \rightarrow S$ `od`, where G is a literal[2], $\neg G$ can be written as an equality $v = w$, where v and w are expressions possibly involving the addition of an introduced constant. This is possible for any loop for which a variant function involving only loop guard variables exists[3]. For such loops, we introduce a constant, C, such that on termination $C = v = w$. A loop can progress toward this state with every iteration only when the difference between v and C or w and C decreases with each iteration of the loop. Limiting ourselves to dealing only with integer variables and constants, and assuming w is greater than v, we present the possibilities for loop progress on a number line in Fig. 7.1.

The sub-figures in Fig. 7.1 are annotated with predicates that describe the changes that must be possible on v and w on successive, arbitrary, iterations of a loop, if the corresponding loops are to make progress. Note, that all of these changes may not actually occur on every iteration, but they must be possible on any iteration. For example, Fig. 7.1(a) states that both v and w must potentially change with every iteration. If the changes to v and w occur in two different paths through the loop body, then both paths must be reachable on an arbitrary iteration.

Figure 7.1 describes the only possible changes allowed to v and w in each iteration, if progress is to be achieved.

Definition 11. General variant Function.

A variant function, t, is called a **general variant function**, if and only if t is a variant function for every loop which has a variant function.

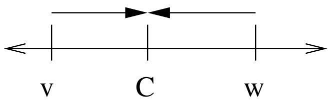
Theorem 7.2. For any loop, $\{Q\}$ `do` $G \rightarrow S$ `od`, where G is a literal and $\neg G$ is equivalent to the equality $v = w$, where v and w are expressions, let C be a constant, such that on termination $C = v = w$. Then $|v - C| + |w - C|$ is a general variant function.

Proof. If we find a function t, such that, for any loop, $0 \leq t_{k+1} < t_k$ holds on any iteration k whenever progress is made toward termination, then t is a general variant function[4].

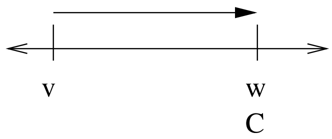
[2] We only consider literals constructed with the predicate symbols $<, \leq, =, >, \geq$ and their negations

[3] Loop guards of the form `do` $i = j$ can have no variant function consisting of only loop guard variables, as the only time the relationship between the loop guard variables changes is on termination.

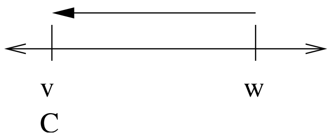
[4] Note we are dealing only with the set of natural numbers as the basis for the well founded set



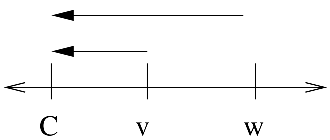
(a) $v_k < w_k \wedge v_k < C < w_k \wedge v_k < v_{k+1} \wedge w_{k+1} < w_k$



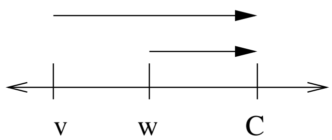
(b) $w_{k+1} = w_k = C \wedge v_k < w_k \wedge v_k < v_{k+1}$



(c) $v_{k+1} = v_k = C \wedge v_k < w_k \wedge w_{k+1} < w_k$



(d) $C < v_k < w_k \wedge v_{k+1} < v_k \wedge w_{k+1} < w_k$ (decreasing toward some constant)



(e) $v_k < w_k < C \wedge v_{k+1} > v_k \wedge w_{k+1} > w_k$ (increasing toward some constant)

Figure 7.1: Possible state changes for loop progress such that on termination $v = w = C$.

Let t be the function $|v - C| + |w - C|$.

For any $k \in \mathbb{N}$, $0 \leq |v_k - C| + |w_k - C|$ holds so, $0 \leq t$ is always true. Therefore, we need only prove that $t_{k+1} < t_k$ holds whenever progress is made.

If the loop is to terminate immediately before some $k + 1$st iteration (at the end of some kth iteration) then, $v_k = w_k = C$ and $|v_k - C| + |w_k - C| = 0$. If a loop does not terminate immediately before an arbitrary $k + 1$st iteration then $v_k \neq C \vee w_k \neq C$, and $0 < |v_k - C| + |w_k - C|$.

$t_{k+1} = |v_{k+1} - C| + |w_{k+1} - C|$ and $t_k = |v_k - C| + |w_k - C|$.

Figure 7.1 describes all of the cases under which progress may be made toward termination. In order for t to be a general variant function, we must show that $t_{k+1} < t_k$ under all of the cases described by Fig. 7.1.

Let w be greater than v.

Case (a): $(v_k < w_k \wedge v_k < C < w_k \wedge v_k < v_{k+1} \wedge w_{k+1} < w_k)$

$$
\begin{array}{cl}
 & t_{k+1} < t_k \wedge (v_k < w_k \wedge v_k < C < w_k \wedge v_k < v_{k+1} \wedge w_{k+1} < w_k) \\
\equiv & \begin{array}{l} |v_{k+1} - C| + |w_{k+1} - C| < |v_k - C| + |w_k - C| \wedge \\ v_k < w_k \wedge v_k < C < w_k \wedge v_k < v_{k+1} \wedge w_{k+1} < w_k \end{array} \\
\equiv & \mathit{true}
\end{array}
$$

Case (b): $(w_k > v_k \wedge v_{k+1} > v_k \wedge w_{k+1} = w_k = C)$

From Fig. 7.1(b), in order for this case to occur, $w_k = C \wedge v_k < C$ must be true.

$$
\begin{array}{cl}
 & t_{k+1} < t_k \wedge (w_k > v_k \wedge v_{k+1} > v_k \wedge w_{k+1} = w_k) \\
\equiv & \begin{array}{l} |v_{k+1} - C| + |w_{k+1} - C| < |v_k - C| + |w_k - C| \wedge \\ w_k > v_k \wedge v_{k+1} > v_k \wedge w_{k+1} = w_k \end{array} \\
\equiv & v_k < C \\
\equiv & \mathit{true}
\end{array}
$$

Case (c): $(w_k > v_k \wedge w_{k+1} < w_k \wedge v_{k+1} = v_k = C)$

Similarly, $t_{k+1} < t_k$ can only be satisfied for this case when $w_k > C$, and this condition is specified by Fig. 7.1(c).

Case (d): $(C < v_k < w_k \wedge v_{k+1} < v_k \wedge w_{k+1} < w_k)$

$$
\begin{array}{cl}
 & t_{k+1} < t_k \wedge (w_k > v_k \wedge v_{k+1} > v_k \wedge w_{k+1} = w_k) \\
\equiv & \begin{array}{l} |v_{k+1} - C| + |w_{k+1} - C| < |v_k - C| + |w_k - C| \wedge \\ w_k > v_k \wedge v_{k+1} < v_k \wedge w_{k+1} < w_k \end{array} \\
\equiv & \mathit{true}
\end{array}
$$

Case (e): $(v_k < w_k < C \wedge w_{k+1} > w_k \wedge v_{k+1} > v_k)$

Similarly, $t_{k+1} < t_k$ is always true for this case.

Because $t_{k+1} < t_k$ holds under all of the state changes described by Fig. 7.1, and these are all of the possible state changes leading to loop progress, then $t_{k+1} < t_k$ always holds when an arbitrary loop progresses. Therefore, $|v - C| + |w - C|$ is a general variant function. □

Lemma 7.1. If either v or w is a constant, then the variant is equivalent to $|v - w|$.

Proof. In the general variant function above, C is a constant that represents the values of the expressions v and w when evaluated at termination, such that, $v = w = C$. Let w be a constant, then by definition, $w_k = C$, for all iterations, k. So $w = C$ always holds and $|v - C| + |w - C| = |v - w| + |w - w| = |v - w|$. Similarly, when v is a constant, then by definition, $v_k = C$, for all iterations, k. So $v = C$ always holds and $|v - C| + |v - C| = |v - v| + |w - v| = |w - v|$.

$|w - v| = |v - w|$, therefore, whenever either v or w is a constant, the variant is equivalent to $|v - w|$ □

Investigating Termination with the General Variant Function.

An attempt to use the general variant function to investigate loop progress by theorem 7.1 reveals its limitations.

Consider again the GCD example (Program 10) under the precondition *true*. Applying theorem 7.1:

Step 1: Calculate the iterative form invariant

Applying (4.1) then (4.2) to the loop body code yields the iterative form invariant

$$R_{k+1} \equiv k \in \mathbb{N} \wedge \left(\begin{array}{l} x_{k+1} = x_k - y_k \wedge y_{k+1} = y_k \wedge x_k > y_k \vee \\ x_{k+1} = x_k \wedge y_{k+1} = y_k - x_k \wedge y_k > x_k \end{array} \right)$$

Step 2: Identify a suitable variant function, t

We have identified the general variant function $|v - C| + |w - C|$ with $v = x$ and $w = y$, so

$$t(x, y) = |x - C| + |y - C| \,.$$

Step 3 and 4: Calculate the progress conditions, PC and simplify by (7.4)

$$\begin{array}{rcl} PC & \equiv & 0 \le t_{k+1} < t_k \wedge R_{k+1} \\ & \equiv & \left[\begin{array}{l} k \in \mathbb{N} \wedge 0 \le |x_{k+1} - C| + |y_{k+1} - C| < |x_k - C| + |y_k - C| \wedge \\ \left(\begin{array}{l} x_{k+1} = x_k - y_k \wedge y_{k+1} = y_k \wedge x_k > y_k \vee \\ x_{k+1} = x_k \wedge y_{k+1} = y_k - x_k \wedge y_k > x_k \end{array} \right) \end{array} \right] \\ & \equiv & k \in \mathbb{N} \wedge \left(\begin{array}{l} y_k < x_k \wedge |x_k - y_k - C| + |y_k - C| < |x_k - C| + |y_k - C| \vee \\ x_k < y_k \wedge |x_k - C| + |y_k - x_k - C| < |x_k - C| + |y_k - C| \end{array} \right) \end{array}$$

Simplifying this to obtain the more useful condition below is a non-trivial exercise, because of the necessity to solve inequalities with absolute values on both sides.

$$\begin{aligned} PC &\equiv k \in \mathbb{N} \wedge (0 < y_k < x_k \vee 0 < x_k < y_k) \\ &\equiv k \in \mathbb{N} \wedge (x_k \neq y_k \wedge 0 < x_k \wedge 0 < y_k) \end{aligned}$$

Due to the complexity of the yielded progress conditions, and the inherent difficulty with simplification, the Variant Conditions Method using the general variant function does not lend itself to automation.

In the next section we describe an approach to deriving progress conditions based on progress rules derived from Fig. 7.1. This approach does not rely on the provision of any variant function, and hence overcomes the limitations of the Variant Conditions Method described in this section. It can be applied directly to the program code and has been automated.

7.3 Investigating Loop Progress Without a Variant

The Variant Conditions Method defined in the previous section is reliant on the identification of a suitable variant function. A variant function is not always available to a verifier, particularly if verifying legacy code. As explained above techniques to derive variant functions are limited in their applicability and the only identified *general variant function* is limited by its complexity. For this reason, we introduce a method in this section for investigating the conditions required for loop progress without reliance on any identified variant function. Important among the advantages of the introduced technique is that it can be completely automated.

The theory is introduced with simple examples where the loop guards are literals. This is extended later. For any loop, $\{Q\}$ `do` $G \rightarrow S$ `od`, where G is an atomic formula, $\neg G$ can be written as an equality $v = w$, where v and w are expressions possibly involving the addition of an introduced constant.

For example, the loop guard $i < j$ when negated gives $i \geq j$. $i \geq j$ can be rewritten as $i = j + C$, where $C \geq 0$. This is in the form $v = w$, with v being i and w being $j + C$.

There are two ways in which a loop may make progress, *with every iteration*, toward achieving a state in which $v = w$. Figure 7.1 illustrates that either the difference between v and w must decrease with each iteration of the loop, or v or w must approach some constant value with every iteration. This constant value is an undefined constant, C, such that on termination, $C = v = w$. It is difficult to reason about termination using rules that involve such a constant. For this reason we attempt to model loop progress by rules that do not involve this constant C.

A technique is presented which yields documentation of the conditions required for loop progress using only information which can be obtained directly from the code. This technique is only accurate for loops where loop progress is possible on *every iteration*. Because we are assuming normalised MBS loops [5], we can be assured that in a correct program this is always the case. The reason for this is that in a normalised MBS loop, all branches are accessible on every iteration [28], so if progress toward termination is to occur, then at least one accessible path through the loop should ensure progress is made on any iteration.

Analysing further the diagrammatic representation of loop progress in figure 7.1 we identify the following conditions that must be met if progress is to be possible on every iteration of a loop. Removing any reference to the constant C, the loop body must ensure that *one* of the following conditions holds on any iteration in order for progress to be achieved:

- the lesser of v and w increases and the greater decreases.
- the lesser of v and w increases and the greater remains constant.
- the lesser of v and w remains constant and the greater decreases.
- the lesser of v and w is non-increasing and the greater is decreasing at a faster rate.
- the greater of v and w is non-decreasing and the lesser is increasing at a faster rate.

Formally we can represent these individual conditions by

$$\begin{aligned}
PC_1 &= (v_k < w_k \wedge v_k < v_{k+1} \wedge w_{k+1} < w_k) \\
PC_2 &= (w_k < v_k \wedge w_k < w_{k+1} \wedge v_{k+1} < v_k) \\
PC_3 &= (v_k < w_k \wedge v_k < v_{k+1} \wedge w_{k+1} = w_k) \\
PC_4 &= (w_k < v_k \wedge w_k < w_{k+1} \wedge v_{k+1} = v_k) \\
PC_5 &= (v_k < w_k \wedge v_k = v_{k+1} \wedge w_{k+1} < w_k) \\
PC_6 &= (w_k < v_k \wedge w_k = w_{k+1} \wedge v_{k+1} < v_k) \\
PC_7 &= (v_k < w_k \wedge v_{k+1} \leq v_k \wedge w_{k+1} < w_k \wedge v_k - v_{k+1} < w_k - w_{k+1}) \\
PC_8 &= (w_k < v_k \wedge w_{k+1} \leq w_k \wedge v_{k+1} < v_k \wedge w_k - w_{k+1} < v_k - v_{k+1}) \\
PC_9 &= (v_k < w_k \wedge v_k < v_{k+1} \wedge w_k \leq w_{k+1} \wedge w_{k+1} - w_k < v_{k+1} - v_k) \\
PC_{10} &= (w_k < v_k \wedge w_k < w_{k+1} \wedge v_k \leq v_{k+1} \wedge v_{k+1} - v_k < w_{k+1} - w_k)
\end{aligned}$$

As with the Variant Conditions Method of the previous section, the iterative-form invariant can be used to assess progress of a loop as it defines $k+1$st iteration instances of a variable in terms of kth iteration instances. So, in order to establish the initial conditions required for loop progress, denoted PC, we

[5] For a discussion of normalised MBS loops see chapter 6 and [28].

can simplify the following for a normalised MBS loop with N paths and an iterative-form invariant R_{k+1}:

$$PC = R_{k+1} \wedge \bigvee_{i=1}^{10} (PC_i) \tag{7.6}$$

As with the previous method, the results apply for any arbitrary iteration. They are, therefore, invariant.

To summarise, we modify theorem 7.1 to use the above progress conditions.

Theorem 7.3. [Progress Conditions Method]

For any loop, $\{Q\}$ `do` $G \rightarrow S$ `od`, apply the following steps:

1. Calculate the iterative form invariant by (4.2).
2. Identify expressions v and w such that on termination $v = w$.
3. Calculate the progress conditions, PC, by (7.6).
4. Simplify the progress conditions, by applying (7.4) to substitute equal expressions for all k+1st iteration variable instances. Drop all k subscripts from the simplified progress conditions.
5. Calculate the failure to progress conditions, FP by negating PC.

Then, PC describes the conditions required for loop progress on any iteration, FP describes the conditions under which the loop will fail to progress toward termination. If the conjunction $Q \wedge FP$ is satisfiable then there are initial conditions allowable under the given precondition that will result in S failing to progress toward termination. The states satisfying $Q \wedge FP$ describe these initial conditions. Conversely, the progress conditions PC, conjoined to the precondition Q, describes the required initial conditions in order for S to progress toward achieving termination with every iteration. $Q \wedge PC$ is, therefore, the weakest precondition, and PC is the weakest invariant required for loop progress on every iteration. Conditions for progress or termination, TC, are defined by $TC = PC \vee \neg G$.

Corollaries 7.1 and 7.2 still hold for the modified theorem.

Consider again the GCD example (Program 10). We apply theorem 7.3 to derive and reason about progress conditions.

Step 1: Calculate the iterative form invariant

Again we assume the weakest possible precondition $true$, ensuring that the loop precondition is $(x, y \in \mathbb{Z} \wedge x = X \wedge y = Y)$. Applying (4.1) then (4.2) to the loop body code yields the iterative form invariant

$$R_{k+1} \equiv k \in \mathbb{N} \wedge \left(\begin{array}{c} x_{k+1} = x_k - y_k \wedge y_{k+1} = y_k \wedge x_k > y_k \vee \\ x_{k+1} = x_k \wedge y_{k+1} = y_k - x_k \wedge y_k > x_k \end{array} \right)$$

Step 2: Identify expressions v and w such that on termination $v = w$

The negation of the loop guard states that on termination the equality $x = y$ holds, so we identify and substitute x for v and y for w.

Step 3 and 4: Calculate the progress conditions, PC, and simplify by (7.4)

$$\begin{array}{rcl}
PC & \equiv & R_{k+1} \wedge \bigvee_{i=1}^{10} (PC_i) \\
& \equiv & \left[\begin{array}{l} k \in \mathbb{N} \wedge \\ \left(\begin{array}{l} \bigvee_{i=1}^{10} (PC_i) \wedge x_{k+1} = x_k - y_k \wedge y_{k+1} = y_k \wedge x_k > y_k \vee \\ \bigvee_{i=1}^{10} (PC_i) \wedge x_{k+1} = x_k \wedge y_{k+1} = y_k - x_k \wedge y_k > x_k \end{array} \right) \end{array} \right] \\
& \equiv & k \in \mathbb{N} \wedge (0 < y_k < x_k \vee 0 < x_k < y_k) \\
& \equiv & k \in \mathbb{N} \wedge (x_k \neq y_k \wedge 0 < x_k \wedge 0 < y_k)
\end{array}$$

Dropping the k subscripts gives the following, more natural, representation of the progress conditions:

$$PC \equiv x \neq y \wedge 0 < x \wedge 0 < y$$

So, whenever the guard is not negated ($x \neq y$) and x and y are both greater than zero at the start of any iteration, the loop will make progress toward termination on that iteration.

This approach has been automated in the prototype (see section B.3). As can be seen, the prototype, even with its basic simplification rules, manages to reduce the twenty disjuncts of $R_{k+1} \wedge \bigvee_{i=1}^{10} (PC_i)$ to a form in which the simplification required is trivially applied manually. That is, the prototype states the progress conditions as $((x < y \wedge 0 < x) \vee (y < x \wedge 0 < y))$, which are equivalent to $x \neq y \wedge 0 < x \wedge 0 < y$, that is, PC.

Step 5: Calculate the failure to progress conditions, FP.

The failure to progress conditions are the negation of the progress conditions. So,

$$\begin{array}{rcl}
FP & \equiv & \neg PC \\
& \equiv & x = y \vee x \leq 0 \vee y \leq 0
\end{array}$$

These progress conditions are equivalent to those derived using the variant function method of the previous section. They therefore have the same implications for reasoning about correctness and reuse.

Automated support has largely been provided for this method by the prototype tool defined by chapter 7 of this thesis. Automation of the method is implemented in two stages.

1. The application of the progress condition rules to the guard written as an equality, achieved through substitution only

2. Simplification of the results

The simplification stage is not fully implemented in the prototype, so some results need simple human simplification to their simplest forms.

This method identifies only the conditions required for progress. It does not consider whether or not the loop guard is strong enough to halt this progress and force termination. The next section presents a method for investigating this.

7.4 Investigating Termination Conditions

In order to completely analyse loop termination, we not only need to reason about loop progress, but also that the guard will at some stage force termination. That is, termination must be investigated based on the principle that in order for a loop to terminate, at some stage the loop guard must be negated.

Consider any loop, $\{Q\}$ `do` $G \rightarrow S$ `od`. In order for termination to be achieved, after some arbitrary kth iteration, where $k \in \mathbb{N}$, $\neg G_k$ must hold. This can be described formally by (7.7).

$$\exists k \in \mathbb{N} \, (\neg G_k) \tag{7.7}$$

In order to evaluate this condition, it is necessary to determine if the loop guard variables can ever be in a state on any single iteration, k, which ensures G is negated.

On some single iteration every program variable in the loop guard must be instantiated with a value that will make the loop guard false, forcing termination. If this can not occur, then termination will not be achieved Obtaining a solution to an iterative-form difference equation for a variable, allows us to define the sequence of values that that variable will be assigned. That is, for a variable v, $v(k)$ is defined to be v_k, where $k \in \mathbb{N}$. The sequence of values that v is instantiated with for the first k iterations, where $k \geq 0$, is given by $\langle v(0), v(1), \ldots, v(k) \rangle$. If, based on iterative-form difference equations in R_{k+1}, a difference equation solution, $v(k)$, exists for all variables, v, in the variable set of the loop guard, G, then the conditions required for termination can be defined by simplifying (7.8).

$$\exists k \in \mathbb{N} \, (\neg G \, [v(k)/v]) \, , \text{ for all } v \in V(G) \tag{7.8}$$

That is, there must be an iteration k, such that the value of every loop guard variable on that iteration forces the guard to be negated.

The method just described is summarised in the following theorem.

Theorem 7.4. [Guard Substitution Method]

For any loop, $\{Q\}$ `do` $G \rightarrow S$ `od`, apply the following steps:

1. Calculate the iterative form invariant by (4.2).

2. Calculate difference equation solutions for all variables in the variable set of the loop guard from the equalities in the iterative form difference equation.

3. Using these difference equation solutions, solve (7.8) for $k \in \mathbb{N}$.

If k is not an element of the set of natural numbers, then the loop will not terminate and $\lceil k \rceil$ is the iteration on which progress toward termination ceases. If k is an element of the set of natural numbers, then (7.8) will simplify to true and the loop will terminate. The k that makes (7.8) true is the number of iterations to termination.

Proof. From the discussion presented above, if (7.8) can not be solved for any natural number iteration count k, then there is no single iteration in which the variables involved in the loop guard will be instantiated with values that will force termination. Because k was chosen arbitrarily, we can generalise and say that termination will never be forced in these situations. Conversely if (7.8) can be solved for at least one natural number k, then there must be at least one iteration k in which the loop guard variables are instantiated with values which negate the loop guard forcing termination. Termination must at some stage occur. □

Corollary 7.3. For any loop, if (7.8) can be solved for k ensuring $k \in \mathbb{N}$, then k is a direct measure of the order complexity of the loop.

Example 7.1. Consider the simple loop:

$\{true\}$
```
i:=0;
do i ≠ 100 →
    i:=i+2;
od
```

Applying the *Guard Substitution Method* results in the following analysis:

Step 1: Calculate the iterative-form invariant, R_{k+1}

The given precondition is *true*. This makes the loop precondition ($i = 0$). The negation of the guard is $i = 100$. We calculate the iterative form invariant, R_{k+1} as

$$R_{k+1} \equiv (k \in \mathbb{N} \wedge i_{k+1} = i_k + 2 \wedge i_k \neq 100)$$

Step 2: Calculate difference equation solutions for all loop guard variables

The only loop guard variable is i. The assignment to i, yields the difference equation $i_{k+1} = i_k + 2$. The solution to this difference equation, is $i(k) = 2k$.

Step 3: Solve (7.8) for $k \in \mathbb{N}$

Simplifying (7.8) to determine the conditions for termination gives:

$$\begin{aligned}
& \exists k \in \mathbb{N}\,((i = 100)\,[i(k)/i]) \\
\equiv\ & \exists k \in \mathbb{N}\,(2k = 100) \\
\equiv\ & \exists k \in \mathbb{N}\,(k = 50) \\
\equiv\ & \mathit{true}
\end{aligned}$$

Step 4: Analyse the result

We can determine that the loop will always progress and terminate under this initialisation as (7.8) simplifies to true.

By corrollary 7.3, we note that the loop will iterate 50 times before termination is forced. ■

This is a simple example that could easily have been reasoned about without this elaborate method. Consider now the example loop in Program 11. This loop is very difficult to reason about without the constructive information yielded by application of the *Guard Substitution Method*.

Program 11 Sum Example

```
{i = I ∧ I ∈ ℝ}
s:=0;
do s ≠ 100 →
    i:=i+1;
    s:=s+i;
od
```

We apply theorem 7.4 to Program 11.

Step 1: Calculate the iterative-form invariant, R_{k+1}

The given precondition asserts $(i = I \wedge I \in \mathbb{R})$. This makes the loop precondition $(i = I \wedge I \in \mathbb{R} \wedge s = 0)$. The negation of the guard is $s = 100$. We calculate the iterative form invariant, R_{k+1} as

$$R_{k+1} \equiv (k \in \mathbb{N} \wedge i_{k+1} = i_k + 1 \wedge s_{k+1} = s_k + i_{k+1} \wedge s_k \neq 100 \wedge I \in \mathbb{R})$$

Step 2: Calculate difference equation solutions for all loop guard variables

The only loop guard variable is s. The assignment to s, yields the difference equation $s_{k+1} = s_k + i_{k+1}$. The assignment to i, yields the difference equation $i_{k+1} = i_k + 1$. The solution to this difference equation, is $i(k) = I + k$. So, the solution to $s_{k+1} = s_k + i_{k+1}$ is

$$\begin{aligned}
s(k) &= \left(\sum_{j=1}^{k} i(j)\right) \\
&= \left(\sum_{j=1}^{k} (I + j)\right)
\end{aligned}$$

Step 3: Solve (7.8) for $k \in \mathbb{N}$

Simplifying (7.8) to determine the conditions for termination gives:

$$
\begin{aligned}
& \exists k \in \mathbb{N}\left((s = 100)\left[s(k)/s\right]\right) \\
\equiv\; & \exists k \in \mathbb{N}\left(\sum_{j=1}^{k}(I + j) = 100\right) \\
\equiv\; & \exists k \in \mathbb{N}\left(kI + \sum_{j=1}^{k}(j) = 100\right) \\
\equiv\; & \exists k \in \mathbb{N}\left(kI + \frac{k(k+1)}{2} = 100\right) \\
\equiv\; & \exists k \in \mathbb{N}^{+}\left(I = \frac{100}{k} - \frac{k+1}{2}\right)
\end{aligned}
$$

Step 4: Analyse the result

From the last line we reason that the loop will only terminate when I is initially a real number that makes $\left(I = \frac{100}{k} - \frac{k+1}{2}\right)$ true, for some natural number k. For example, we can reason by substituting $k = 1, 2,$ or 3 that the loop will terminate when $I = 99$, $I = 48.5$, or $I = 31\frac{1}{3}$, respectively. By corollary 7.3, we reason that the loop terminates in a number of iterations equal to the positive roots of the equation

$$\left(I = \frac{100}{k} - \frac{k+1}{2}\right)$$

■

This process is powerful, not only for deriving these conditions but also for testing theories relating to termination. For example, consider the sum example above with the initialisation `i:=0;`. Does the loop terminate under these conditions? While testing such a theory provides an answer immediately, the above method will produce an answer as well as constructive information describing the state when progress toward termination failed. This example is worked below.

Step 1: Calculate the iterative-form invariant, R_{k+1}

The initialisation `i:=0;` makes the loop precondition $(i = 0 \wedge s = 0)$. The negation of the guard is $s = 100$. We calculate the iterative form invariant, R_{k+1} as

$$R_{k+1} \equiv (k \in \mathbb{N} \wedge i_{k+1} = i_k + 1 \wedge s_{k+1} = s_k + i_{k+1} \wedge s_k \neq 100)$$

Step 2: Calculate difference equation solutions for all loop guard variables

The only loop guard variable is s. The assignment to s, yields the difference equation $s_{k+1} = s_k + i_{k+1}$. The assignment to i, yields the difference equation

$i_{k+1} = i_k + 1$. The solution to this difference equation, is $i(k) = k$. So, the solution to $s_{k+1} = s_k + i_{k+1}$ is

$$\begin{aligned} s(k) &= \left(\sum_{j=1}^{k} i(j)\right) \\ &= \left(\sum_{j=1}^{k} j\right) \\ &= \frac{k(k+1)}{2} \end{aligned}$$

Step 3: Solve (7.8) for $k \in \mathbb{N}$

Simplifying (7.8) to determine the conditions for termination gives:

$$\begin{aligned} & \exists k \in \mathbb{N}\,((s = 100)\,[s(k)/s]) \\ \equiv\ & \exists k \in \mathbb{N}\left(\frac{k(k+1)}{2} = 100\right) \\ \equiv\ & \exists k \in \mathbb{N}\left(\frac{k^2+k}{2} = 100\right) \\ \equiv\ & \exists k \in \mathbb{N}\left(k^2 + k = 200\right) \\ \equiv\ & false \end{aligned}$$

Step 4: Analyse the result

Because (7.8) simplified to $false$, we can reason that s will never equal 100 in any whole number of iterations. Therefore, under this initialisation, the guard will never be negated and the loop will never terminate. Solving the quadratic equation on the second last line of the above gives $k = 13.65$ (k can't be negative so we are only concerned with positive roots). We can reason from this, that up to the 13th iteration, the loop is making progress toward the termination state. That is $s < 100$ and s is increasing toward 100. However, the guard is not negated, and after iteration 14, the next whole number iteration, $s > 100$, and the loop will fail to progress. ■

7.4.1 Comparison with Variant Techniques

This is a useful method for calculating all progress and termination conditions for loops where all loop guard variables are associated with solvable difference equations in R_{k+1}. It can be applied as an independent method or as part of a difference equation approach to calculating the loop invariant.

This technique is reliant on the ability to solve the iterative-form assertions in the iterative-form invariant for only the loop guard variables. This is possible in many well-structured programs. This technique is applicable in all of the cases that the well-founded-sets methods of [34, 72] can be applied. These techniques are reliant on identification of an invariant in order to extract a suitable variant function to prove that the termination conditions are true. This is due to the fact that for a variant function to be suitable, it must be implied by the invariant. If this can't be proved, then a verifier can not be sure that the variant function being used for termination analysis is suitable. In order to use these methods, all paths through the program must yield the same invariant with respect to the loop guard variables. This is the case with variant-based techniques also [72]. The method described by Theorem 7.4, has the benefit that it describes a technique for repeatably extracting *constructive* information describing the conditions required for termination.

This technique lends itself to automation. It is not reliant on generating an entire invariant, or on the identification of a sufficient variant function. The generation of iterative form invariants has already been automated in a prototype as have the solutions to many commonly occurring difference equations. The third step above, the solution and simplification of (7.8) has not been fully automated in the prototype at this stage. It is expected that the automation of solving and simplifying (7.8) when applied to code examples like the sum example above is non-trivial.

The limitations of this method have been described above. Important among these is the fact that this technique requires the derivation of invariant relationships, in non-iterative form, for all loop guard variables. This may not always be possible when, for example, the iterative-form difference equations are difficult to solve or not solvable, or different paths through the loop progress toward termination by differing magnitudes. In these situations it is always possible to establish the conditions required for loop progress by the *Progress Conditions Method* of the previous section.

7.5 Other Loop Guard Forms

7.5.1 Guards With = Connectives

In order to reason about termination when dealing with loops of the form `do` $v = w \rightarrow S$ `od`, we can make use of the principle introduced in the previous section[6]. That is, if termination is to be achieved, then the negation of the guard must be forced on some arbitrary kth iteration. That is, the following must hold:

$$\exists k \in \mathbb{N}\,(\neg G_k) \tag{7.9}$$

Therefore, we can analyse the progress and termination characteristics of this class of loop only when the difference equations in the iterative-form invariant, R_{k+1} can be solved for all of the loop guard variables. So termination and progress of such loops can only be analysed when the *Guard Substitution*

[6]The appearance of loop guards with = as the main connective is often an indicator of an improperly formulated loop.

Method is applicable, otherwise we can only present an iterative form condition corresponding to (7.9)

For example, consider the loop guard `do` $i = j + 1 \rightarrow$. If $i(k)$ and $j(k)$ can be calculated, then the progress and termination conditions can be found by theorem 7.4, by simplifying

$$\exists k \in \mathbb{N}\,(i(k) \neq j(k) + 1)$$

If $i(k)$ can not be calculated or $j(k)$ can not be calculated then the only statement we can make about termination conditions is

$$\exists k \in \mathbb{N}\,(i_k \neq j_k + 1)$$

Progress can be assessed for such loops by theorem 7.3, realising that on termination

$$i_k + C = j_k + 1 + D \wedge C \neq D$$

7.5.2 Loops with Complex Guards

So far we have only discussed the analysis of loops with simple, literal guards. In order to apply these methods to any real examples, it must be applicable to loops with complex guards. Here we define a complex guard as being a conjunction or disjunction of quantifier free formulae.

Theorem 7.5 (Complex Guards). For any loop with a complex guard, G, for each literal, p_i, in $\neg G$, apply the method described in this section to derive the progress conditions for that part of the guard. If the progress conditions for p_i are denoted by PC_i, then the progress conditions for the entire loop will be given by

$$PC \equiv \neg G\,[PC_i/p_i]\,,\ \text{for all } p_i \text{ in } G \tag{7.10}$$

Proof. If the negated guard is conjunctive, progress must be made toward achieving each conjunct. That is, given a negated guard of the form $p_1 \wedge \ldots \wedge p_n$, let $PC_1 \ldots PC_n$ be the conditions required for progress toward achieving $p_1 \ldots p_n$ respectively. Then, in order for the loop to progress toward termination the loop must progress toward achieving all of the conjuncts in the conjunctive termination condition. If any p_i is not achieved then termination will not succeed. So, the conditions required for progression toward a conjunctive termination condition are the conjunction of the individual conjunct's progress conditions, $PC_1 \wedge \ldots \wedge PC_n$.

If the negated guard is disjunctive, progress must be made toward achieving at least one of the disjuncts of the negated guard. That is, given a negated guard of the form $p_1 \vee \ldots \vee p_n$, let $PC_1 \ldots PC_n$ be the conditions required for progress toward achieving $p_1 \ldots p_n$ respectively. Then, in order for the loop to progress toward termination the loop must progress toward achieving at least one of the disjuncts in the disjunctive termination condition. If no p_i is achieved then termination will not succeed. So, the conditions required for progression toward a disjunctive termination condition are the disjunction of the individual disjunct's progress conditions, $PC_1 \vee \ldots \vee PC_n$.

This is recursive for each sub-formula of the negated guard. $\square$

For example, consider the following:

$\{i = I \wedge j = J\}$
```
do i≠100 ∧ j≠0 →
    i:=i+1; j := j-1;
od;
```

The termination condition is $i = 100 \vee j = 0$ The conditions for progress toward achieving $i = 100$ are derived as $i < 100$. The conditions for progress toward achieving $j = 0$ are derived as $j > 0$. Therefore in order for the loop to progress toward termination, either $i < 100$ must hold, or $j > 0$ must hold, on every iteration. If this is not the case progress is made toward neither disjunct of the termination condition.

By (7.10), the progress conditions are

$$i < 100 \vee j > 0.$$

The failure to progress conditions are, therefore,

$$i \geq 100 \wedge j \leq 0.$$

The given loop precondition does not exclude these states, so it is possible to begin this loop with initial values that will cause progress failure.

By corollary 7.2, these defects could be corrected in a number of ways. The loop precondition could be strengthened to

$$(i \leq 100 \vee j \geq 0 \wedge i = I \wedge j = J)$$

Alternatively the guard could be strengthened to

```
do i<100 ∧ j>0 →
```

7.6 Using the Derived Conditions

The derived failure to progress and termination failure conditions can obviously be used to assess whether or not the program being inspected will progress and terminate under all of the states described by the given precondition. The following criteria would form a good candidate addition to an inspection checklist.

"Are there loop failure to progress and terminate conditions that are achievable under the given precondition?"

This question would have been difficult to assess through inspection prior to the introduction of the methods of this chapter. Applying these methods to derive and analyse the conditions under which loop progress and termination fails, ensures that the question can easily be answered by anyone who can interpret the given precondition and derived invariant.

As implied by corollary 7.2, not only are these conditions useful for reasoning about correctness, they are also constructive in that they can be used to support the correction of termination related defects. Defect correction is discussed in detail below.

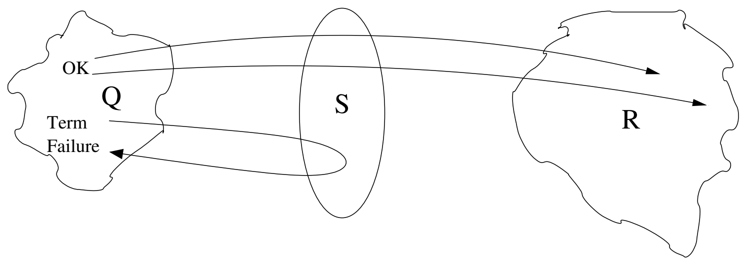
Summarising corollary 7.2, Fig. 7.2 illustrates the different uses of the derived termination information.

Defect Correction by Strengthening the Precondition. Considering the loop, `do` $G \rightarrow S$ `od`, executing under the precondition Q. The progress conditions, PC, describe the states required for S to progress toward termination. The conjunction of the given precondition and the progress conditions, $Q \wedge PC$, describes the weakest precondition required for S to always progress. PC is also an invariant of S. That is PC must be maintained on every iteration if progress is to be achieved. In this respect, by definition, it is not strictly an invariant, as it does not hold on termination. Based on this, we can therefore correct any failure to terminate defects by strengthening the precondition to $Q \wedge PC$, or, for languages that allow assertions, by asserting PC, as an invariant, in the body of the loop.

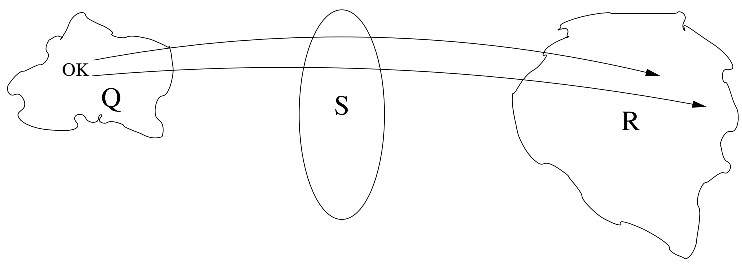
Modifying the Loop. In some cases the given precondition may be correct although the derived failure to progress conditions are allowed by it. In these cases, the loop itself must be defective. That is, the loop guard or loop body statements are responsible for failure to progress and terminate. In such cases, we may not be permitted to strengthen the given precondition to $Q \wedge PC$. In these cases we need to modify the loop so that progress is achieved under the given precondition, Q. There are basically two ways in which we can modify the loop itself to progress under the given precondition.

Firstly, we can strengthen the loop guard. For any loop, $\{Q\}$ `do` $G \rightarrow S$ `od`, strengthening the guard to $G \wedge PC$, will ensure that progress is always made by S under the precondition Q. For example, if we strengthen the guard in the GCD example to be, `while (x!=y && x>0 && y>0)`, progress is always made under the precondition $(x = X \wedge y = Y \wedge x, y \in \mathbb{Z})$. This is logically the same as asserting PC as an invariant at the beginning of each iteration.

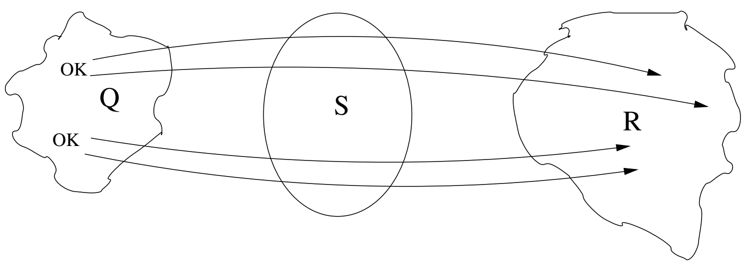
In many cases, we will not wish to strengthen the guard in this way. Strengthening the guard will ensure termination, however, it may not ensure the correct invariant is maintained, and therefore it may cause the loop to terminate with an incorrect postcondition. In such cases, we need to modify the body of the loop itself and possibly also the guard, so that progress may be achieved while



(a) Non progression defects occur when the precondition allows a program to be executed under states that cause a loop to iterate indefinitely



(b) Progression defects can be corrected by strengthening the precondition to restrict the execution of a program to only those states that cause progression.



(c) Progression defects can also be corrected by modifying the program to progress under all states described by the original precondition.

Figure 7.2: A state space description of termination defects and their correction

Program 12 A Defective Binary Search Algorithm

```
i = 0; j = N;
do {
    p = (i+j)/2;
    if (a[p] < x) {
        i = p;
    }
    else if (a[p] > x) {
        j = p;
    }
} while (a[p]!=x && i<j);
```

maintaining the correct invariant. The failure to progress conditions, FP, may be of benefit in guiding the re-engineering of such loops.

Consider the defective binary search in Program 12. The loop progress calculations for this example yield the following conditions

$$\begin{array}{rcl} PC & \equiv & (a\,[(i+j)\,div2] \geq x \vee j \neq i+1) \\ FP & \equiv & (a\,[(i+j)\,div2] < x \wedge j = i+1) \end{array}$$

It does not make sense to modify the precondition of this loop to state that for all i and j, $a\,[(i+j)\,div2] < x \implies j \neq i+1$. Modifying the guard, by conjoining $a[(i+j)/2] \geq x \vee j \neq i+1$ to the existing guard to give: `while ((a[p]!=x && i<j-1)||(a[p]>x && i<j))`, raises a problem with partial correctness. If $i = j-1$ and $a[i] < x$, $a[j]$ will not be checked and $a[j] = x$ may hold. We could modify the program by adding a check for $a[j] = x$ after the loop. Such a solution would be correct, however, the information provided by the method would be better used to guide the re-engineering of the entire loop for a more elegant and maintainable solution.

7.7 Conclusions

In this chapter we have presented repeatable and automatable methods for deriving the conditions which cause loop progress and termination failure. We have demonstrated the application of the methods and discussed the use of their results in assisting the re-engineering and documentation of loops. These methods are not dependent on the existence of any formal documentation for the loop being analysed. As such, they are useful for documenting and re-engineering loops in legacy programs for which existing formal methods are not applicable.

The methods presented form a basis for investigating termination in a human reasoning based code inspection process. The progress conditions method and termination conditions methods have both been implemented in the current prototype, however, the simplification module of the prototype is immature. For example, for the GCD example, the tool generates the following report:

Loop Progress Conditions

The loop will progress in an iteration, when the following holds at commencement of that iteration:

$$\left(\begin{array}{l} (x < y \wedge 0 < x) \vee \\ (y < x \wedge 0 < y) \end{array}\right)$$

Loop Termination Conditions

The loop will terminate, when the following is satisfiable:

$$\exists k \in \mathbb{N}\,(x_k = y_k)$$

Note that the progress conditions are not expressed in their simplest form and the loop termination conditions are expressed iteratively because their were no solvable difference equations for x and y. Work is continuing on extending the simplification module of the prototype.

Chapter 8

Procedure Semantics

8.1 Introduction

In order to trust software so that it may be safely and correctly reused, the components to be reused must be specified in terms of a given precondition and postcondition and must be shown to be correct with respect to the given specification. The precondition must unambiguously state the conditions under which that component can be executed and guarantee to satisfy the postcondition. This is known as programming by contract [74]. In many legacy systems such a specification simply does not exist.

Previous chapters have focussed on the use of the strongest postcondition predicate transformer for deriving specifications from self-contained units of code, that is code that does not call procedures. In this chapter, we present techniques and algorithms for using the strongest postcondition for deriving specifications directly from code for procedures and for deriving specifications of the semantic effects of calling those procedures. In order to reason about the semantic effects of a procedure call, the semantic specification of the procedure being called needs to be abstracted independent of the remainder of the program. That is, the semantic definition of a procedure needs to be abstracted with respect to the formal parameters and other variables that are accessible to the procedure in and of itself. This abstract procedural specification is then manipulated by substitution to describe the effect of a specific procedure call with specific values assigned to the parameters and other accessible variables.

An alternate option is to abstract the initial conditions that hold at invocation of a procedure, given a specific procedure call, and use these conditions as a precondition to the procedure body to calculate its strongest postcondition. Such an approach is suggested in [47, 38] for calculating the semantic effects of procedure calls to non-recursive procedures. As this approach is dependent entirely on the actual parameters being passed, it means that the strongest postcondition of the procedure body must be computed for every procedure call.

The initial approach allows the semantics of a procedure to be calculated once for the procedure as an abstract unit. The semantics of the procedure can then be stored so that whenever a procedure call is made to that procedure, calculation of the procedure semantics is not required. Instead, we simply apply actual for formal parameter substitutions in the stored semantic specification for a procedure to define the semantics of a call to that procedure. In making these substitutions certain types of defects are automatically detectable. This chapter addresses both the abstraction of procedure semantics and the computation of procedure call semantics. The approach described in this chapter is incapable of calculating of specifications from recursive procedures and calls to such procedures. This is also a shortcoming of the related approaches in [47, 38]. For a treatment of the semantics of recursive procedures we refer the reader to [54].

8.2 Procedures and Procedure Call Principles

Consider a procedure call

```
...
{Pre}
proc(al);
```

al is a parameter list comprising zero or more *actual parameters.*

This procedure call is to `proc`, defined by

```
{Q}
procedure proc(pl) {
    S
}
```

where *pl* is a parameter list comprising zero or more formal parameters. Each formal parameter is defined by a parameter declaration of the form (in EBNF):

$$[\mathbf{IN} \mid \mathbf{OUT} \mid \mathbf{INOUT}]\, p : T$$

where p is the name of a *formal parameter* and T is the type of the formal parameter.

There are three types of formal parameter in our extension to the guarded commands language[1]:

- `IN` parameters, or *value parameters*, which are passed by value from the invoking unit to the procedure at the time the procedure is called. The value of the actual parameter is passed to the formal parameter at the time of procedure call

[1] It should be noted that in the language handled by the prototype tool, all parameters are considered to be *reference* parameters.

- `OUT` parameters, or *result parameters*, which are passed from the procedure to the invoking unit at the time of return. The value of the formal parameter is assigned to the actual parameter at the time of procedure return.

- `INOUT` parameters, or *reference parameters*, which pass information both ways between the invoking unit and the procedure. The address, or reference, to the actual parameter is passed to the procedure and the formal parameter refers to this address. Hence, any manipulation of the formal parameter by the procedure is a manipulation of the actual value in the invoking unit.

The semantics of each type of parameter are different. We treat each separately below in defining the strongest postcondition of a procedure, and procedure call.

In the following section we investigate how to abstract the semantics of a procedure. We then investigate how these semantics can be used to derive the strongest postcondition of a procedure call.

8.3 Abstracting the Semantics of a Procedure

If we restrict our analysis to procedures that have no side-effects, then the semantics of a procedure can be determined by examining the state changes of the variables accessible to the procedure. These variables include the formal parameters of the procedure.

To describe the state changes made by a procedure to its formal parameters, the initial values of the formal parameters must be accessible. Considering a procedure as an abstract unit of code, the details of the *actual parameters* passed to the procedure are hidden from the procedure until invocation. Therefore, the semantics of a procedure must be represented in terms of any allowed actual parameters. This is also the case for all other variables accessible and used within the procedure that are declared and defined beyond the scope of the procedure. A procedure referencing or updating global variables, or a method referencing or updating class variables is an example of this. Here we concentrate on defining formal parameters in terms of all possible associated actual parameters.

8.3.1 Formal Parameter Semantics

In our language, as with most imperative, non-scripting languages, all formal parameters are typed. That is, each formal parameter is declared in the parameter list as being of a particular type. The type of a formal parameter constrains the values that the parameter variable may assume. If a formal parameter p is declared to be of type T, then $p \in T$ must hold.

A positional parameter association mechanism has been chosen. So, in order to facilitate association, parameters must be referenced within a parameter list by their position in that list. The following is a notation to allow us to discuss parameter positions:

Notation. For any parameter list pl the notation pl_i is used to refer to the name of the parameter at position i in pl and $TYPE_i$ refers to the type of the parameter at position i in pl.

Consider, for example, the parameter list $pl =$ (`IN x: Type, OUT y: Type`), then pl_1 refers to the parameter x and pl_2 refers to the parameter y.[2]

For all `IN`, and `INOUT` type parameters, the formal parameter is instantiated with a value when the procedure is invoked. Note that it is unimportant, at this stage, that this instantiation is by reference for INOUT type parameters. We borrow Z notation [87] to represent the initial values associated with a procedure's formal parameters. For every formal value or reference parameter pl_i, in a formal parameter list, pl, we represent the initial value of pl_i, abstractly, by $pl_i?$.

Result, or `OUT`, parameters are not initialised by invocation, and are therefore only constrained by type by the procedure declaration.

Now, for every formal *value* or *reference* parameter, pl_i, declared in the parameter list of a procedure, where pl_i is of type T, the initial constraints on pl_i imposed by the procedure declaration are described by

$$pl_i = pl_i? \wedge pl_i \in T \tag{8.1}$$

For every formal *result* parameter, pl_i, , where pl_i is of type T, the initial constraints on pl_i imposed by the procedure declaration are described by

$$pl_i \in T \tag{8.2}$$

A procedures parameters may also be constrained by the precondition of the procedure. For example, consider the following procedure:

$\{x > 0\}$
`procedure sqrt(IN x: ℤ, OUT y: ℝ) { ... }`

The formal value parameter x is constrained by declaration to the set of integers and by the precondition to be greater than 0. So, we identify the initial conditions on x as $x = x? \wedge x \in \mathbb{Z} \wedge x > 0$.

The initial conditions for any procedure, `proc(`pl`)`, denoted Q_{proc}, under the precondition Q, are defined by (8.3).

Let S_{vr} represent the set of positions of all value and reference parameters in the parameter list, pl, and let S_{res} represent the set of positions of all result parameters in the parameter list, pl, then

$$Q_{proc} \equiv \bigwedge_{i \in S_{vr}} (pl_i = pl_i? \wedge pl_i \in TYPE_i) \wedge \bigwedge_{i \in S_{res}} (pl_i \in TYPE_i) \wedge Q \tag{8.3}$$

[2]We may abbreviate a parameter list. For example **procedure proc(IN x,y: String)** is equivalent to **procedure proc(IN x: String, IN y: String)**.

The semantics of a procedure can now be defined by application of the strongest postcondition predicate transformer.

Let Q_{proc}, calculated by (8.3), be the initial conditions of a procedure, `proc`, with procedure body S, then the postcondition of the procedure, denoted R_{proc}, is given by

$$R_{proc} \equiv sp\,(Q_{proc},\ \mathtt{S}) \tag{8.4}$$

For readability, we may wish to remove references to local procedure variables from the procedure postcondition. This can be achieved by exploiting the substitution property of equality. For any local variable v in the variable set of the postcondition, R_{proc}, of a procedure, `proc`, if $R_{proc} \implies v = E$, where E is an expression, then

$$R_{proc} \equiv R_{proc}[E/v]. \tag{8.5}$$

Program 13 A procedure to add to integer Vectors

$\{\#v = \#w\}$

```
procedure add(IN v,w:  seq ℤ, OUT sum:  seq ℤ ) {
    i := 0;
    do i < #v →
        sum[i] := v[i]+w[i];
        i := i+1;
    od
}
```

Consider the `add` procedure in Program 13. This procedure is intended to add two given vectors and store the result in a third vector.

By (8.3), the initial conditions of the `add` procedure are

$$Q_{add} \equiv (\#v = \#w \wedge v = v? \wedge v \in \mathtt{seq}\,\mathbb{Z} \wedge w = w? \wedge w \in \mathtt{seq}\,\mathbb{Z} \wedge sum \in \mathtt{seq}\,\mathbb{Z})$$

Applying the strongest postcondition predicate transformer to the procedure body and the precondition above, gives the following postcondition for the `add` procedure, denoted R_{add}.

$$\begin{array}{rcl} R_{add} & \equiv & sp\,(Q_{add},\ \mathtt{S}_{add}) \\ & \equiv & \left(\begin{array}{l} \#v = \#w \wedge v = v? \wedge w = w? \wedge v \in \mathtt{seq}\,\mathbb{Z} \wedge w \in \mathtt{seq}\,\mathbb{Z} \wedge \\ sum \in \mathtt{seq}\,\mathbb{Z} \wedge \forall j \in [0..i-1]\,(sum[j] = v?[j] + w?[j]) \wedge i = \#v \end{array} \right) \end{array}$$

By (8.5), we would remove the references to the local variable i in R_{add} as follows:

$$\begin{array}{rcl} R_{add} & \equiv & \left(\begin{array}{l} \#v = \#w \wedge v = v? \wedge w = w? \wedge v \in \mathtt{seq}\,\mathbb{Z} \wedge w \in \mathtt{seq}\,\mathbb{Z} \wedge \\ sum \in \mathtt{seq}\,\mathbb{Z} \wedge \forall j \in [0..i-1]\,(sum[j] = v?[j] + w?[j]) \wedge i = \#v \end{array} \right)[\#v/i] \\ & \equiv & \left(\begin{array}{l} \#v = \#w \wedge v = v? \wedge w = w? \wedge v \in \mathtt{seq}\,\mathbb{Z} \wedge w \in \mathtt{seq}\,\mathbb{Z} \wedge \\ sum \in \mathtt{seq}\,\mathbb{Z} \wedge \forall j \in [0..\#v-1]\,(sum[j] = v?[j] + w?[j]) \end{array} \right) \end{array}$$

In effect this section describes a method for the derivation of a procedure's implied postcondition in terms of its given precondition. This semantic information can be used to reason about a procedure's correctness. Chapter 9 discusses in detail techniques for the verification of procedures.

We summarise the technique discussed in this section in algorithm 12.

Algorithm 12 Abstracting Procedure Semantics

Consider the procedure `proc(`*pl*`)`, with the precondition Q, where *pl* is a formal parameter list. Calculate the abstract semantics of `proc` by the following steps:

1. Calculate Q_{proc}, the initial conditions for *proc*, by (8.3).
2. Calculate R_{proc}, the strongest postcondition of `proc` under Q_{proc}, by (8.4).
3. Remove references to local variables in R_{proc} by (8.5).

In the next section we investigate the use of procedure postconditions to describe the semantics of procedure calls.

8.4 Procedure Call Semantics

8.4.1 Parameter Substitution

In order to describe a method for calculating the strongest postcondition of a procedure call in terms of the called procedure's derived postcondition, it is necessary to provide a mechanism for substituting *actual* for *formal* parameters in the derived postcondition of the called procedure.

A positional association between formal and actual parameters ensures that each formal parameter in a procedure declaration is associated with the value parameter in the corresponding parameter position in a procedure call to that procedure. For example, consider the following procedure declaration:

```
procedure testProc(a:  IN ℤ, b:  INOUT ℤ)
```

Then a call of `testProc` associating the actual parameter `x+y` with formal value parameter `a`, and the actual parameter `y` with formal reference parameter `b`, would be:

```
testProc(x+y,y);
```

Each type of parameter requires a different mechanism to substitute actual for formal parameters in the procedure postcondition. We consider each separately now[3]. For a procedure `proc` with the postcondition R_{proc}, we denote the post-

[3]For this discussion we ignore issues of the mechanics of memory allocation when describing procedures. For such a discussion, any text discussing imperative programming language principles should suffice.

condition of `proc` with actual parameters substituted for formal parameters by R'_{proc}.

Value (`IN`) Parameters

A value parameter is a formal parameter into which the value of the associated actual parameter is copied when a procedure is called. Any changes made by the procedure to the formal value parameter do not effect the associated actual parameter in the calling context.

Let a, be an actual parameter associated with a formal value parameter, p, of a procedure, `proc`. This association is semantically equivalent to assigning a to p at procedure invocation. Assigning a to p prior to execution of the procedure body yields the equality $p = a$ prior to execution. By (8.1), we already know that for the formal value parameter p, $p = p?$ holds prior to execution of the procedure body. As $p = a$ and $p = p?$ both hold prior to execution of the procedure body, then $p? = a$ must hold prior to execution of the procedure body. Neither $p?$ or a are accessible to the procedure body, as p is a value parameter. Therefore, $p?$ and a are not changed by the procedure, and $p? = a$ always holds. So, substituting a for $p?$ in R_{proc} represents the semantics of the call to `proc`, where actual parameter a is associated with formal value parameter p.

If a procedure, `proc`(pl), is called with `proc`(al), where pl and al are formal and actual parameter lists respectively, then for every formal **value** parameter pl_i in pl and corresponding actual parameter al_i in al, the following equality allows us to describe the procedure postcondition of `proc` in terms of the actual parameters:

$$R'_{proc} \equiv R_{proc}\,[al_i/pl_i?] \tag{8.6}$$

Result (`OUT`) Parameters

A result parameter is a formal parameter which is bound to any element of its type at invocation. That is, initially, the value of a formal parameter, p, is only known to be some value of the type of p. An actual parameter, a, associated with a formal result parameter p must be a *reference* to a variable. At invocation, p becomes bound to a, and any changes to p made by the procedure are actually made to a. So, $a = p$ holds at every stage of the procedure. Therefore, on exit from the procedure, $a = p$ holds.

Let a, be an actual parameter associated with a formal result parameter, p, of a procedure, `proc`. This association implies that the equality $a = p$ is maintained on exit from the procedure. R_{proc} describes the state of the procedure, `proc`, on exit. So, substituting a for p in R_{proc} represents the semantics of the call to `proc`, where actual parameter a is associated with formal result parameter p.

If a procedure, `proc`(pl), is called with `proc`(al), where pl and al are formal and actual parameter lists respectively, then for every formal **result** parameter

pl_i in pl and corresponding actual parameter al_i in al, the following equality allows us to describe the procedure postcondition of `proc` in terms of the actual parameters:

$$R'_{proc} \equiv R_{proc}\,[al_i/pl_i] \tag{8.7}$$

Reference Parameters (`INOUT`)

The reference parameter mechanism is a combination of the mechanisms for value parameters and result parameters. For any procedure, `proc`, a formal reference parameter p associated with an actual parameter a is initially bound to the reference variable a. So initially, the value referred to by p is equivalent to the value referred to by a, and a and p are equivalent on exit.

Let $a = e$ hold on invocation of `proc`, where e is some expression. This association semantically implies the equalities $p? = e$ and $a = p$ on invocation. These equalities are maintained throughout the procedure, as $p?$ and e are not accessible to the procedure, and p is an alias for a. R_{proc} describes the state of `proc` on exit. So, substituting e for $p?$ and a for p in R_{proc} represents the semantics of the call to `proc`, where actual parameter a is associated with formal reference parameter p.

If a procedure, `proc`(pl), is called with `proc`(al), where pl and al are formal and actual parameter lists respectively, then for every formal **reference** parameter pl_i in pl and corresponding actual parameter al_i in al, where $al_i = e$ and e is an expression, the following equality allows us to describe the procedure postcondition of `proc` in terms of the actual parameters:

$$R'_{proc} \equiv R_{proc}\,[e, al_i/pl_i?, pl_i] \tag{8.8}$$

Following these substitutions, all remaining references to formal parameters can be removed by applying (8.5) to R'_{proc}.

As an example, consider the following procedure call to `add` in Program 13:

$$\{a = a? \land b = b? \land s \in \texttt{seq}\,\mathbb{Z}\}$$
```
add(a,b,s);
```

The postcondition for the procedure `add` has been abstracted above, and is known to be

$$R_{add} \equiv \left(\begin{array}{l} \#v = \#w \land v = v? \land w = w? \land v \in \texttt{seq}\,\mathbb{Z} \land w \in \texttt{seq}\,\mathbb{Z} \land \\ sum \in \texttt{seq}\,\mathbb{Z} \land \forall j \in [0..\#v-1]\,(sum[j] = v?[j] + w?[j]) \end{array}\right)$$

The formal parameters v and w are value parameters. v is associated with the actual parameter a by the procedure call so $v? = a$ holds. Similarly w is associated with b and $w? = b$ holds. The formal parameter sum is a result parameter associated with the reference variable s.

Applying (8.6) for the value parameters v and w and (8.7) for the result parameter sum we refine the procedure postcondition R_{add} to the following, expressed in terms of actual parameter variables and values:

$$\begin{aligned} R'_{add} &\equiv \left(\begin{array}{l} \#v = \#w \land v = v? \land w = w? \land v \in \texttt{seq}\,\mathbb{Z} \land w \in \texttt{seq}\,\mathbb{Z} \land \\ sum \in \texttt{seq}\,\mathbb{Z} \land \forall j \in [0..\#v-1]\,(sum[j] = v?[j] + w?[j]) \end{array} \right) [a, b, s/v?, w?, sum] \\ &\equiv \left(\begin{array}{l} \#v = \#w \land v = a \land w = b \land v \in \texttt{seq}\,\mathbb{Z} \land w \in \texttt{seq}\,\mathbb{Z} \land \\ s \in \texttt{seq}\,\mathbb{Z} \land \forall j \in [0..\#v-1]\,(s[j] = a[j] + b[j]) \end{array} \right) \end{aligned}$$

We can now apply (8.5), to remove the references to the formal parameters as they correspond to local variables in R'_{add}. This yields the following:

$$R'_{add} \equiv \left(\begin{array}{l} \#a = \#b \land a \in \texttt{seq}\,\mathbb{Z} \land b \in \texttt{seq}\,\mathbb{Z} \land \\ s \in \texttt{seq}\,\mathbb{Z} \land \forall j \in [0..\#v-1]\,(s[j] = a[j] + b[j]) \end{array} \right)$$

Detecting Repeated Actual Reference Parameter Defects

As a side effect of reference parameter substitution (8.8), defects relating to repeating actual reference parameters are made apparent by a resulting contradiction. As an example, consider the procedure call $\{a = 2\}$`aproc(a,a)` to a procedure `aproc(INOUT x,y :` $\mathbb{Z}$`)` with the derived postcondition $R_{aproc} \equiv (x = 2y? \land y = 3x? \land x, y \in \mathbb{Z})$. Applying (8.8) gives

$$\begin{aligned} R'_{aproc} &\equiv \left(\begin{array}{l} x, y \in \mathbb{Z} \land \\ x = 2y? \land y = 3x? \end{array} \right) \begin{array}{l} [2, a \,/\, x?, x] \\ [2, a \,/\, y?, y] \end{array} \\ &\equiv (a, a \in \mathbb{Z} \land a = 2 * 2 \land a = 3 * 2) \\ &\equiv (a \in \mathbb{Z} \land a = 4 \land a = 6) \\ &\equiv false \end{aligned}$$

This gives rise to the following inspection rule:

Inspection Rule 4 (Detect Repeated Actual Reference Parameters). After applying (8.8) in calculating the postcondition of a procedure call to a procedure `proc`, answer the following question:

> "Does R'_{proc} contain contradictions as a result of substitution (8.8)?"

8.4.2 Checking Procedure Preconditions

The design by contract [74] paradigm promotes correctness by describing a procedure as having a contract. All procedures have a precondition and a postcondition. The client, or caller, of a procedure is entitled to expect that if the precondition of the called procedure is met, then the procedure is guaranteed to terminate and result in a state that satisfies the postcondition. The client is, therefore, obligated to ensure that the precondition is met. On the other hand, the procedure itself is entitled to expect that the precondition is met and is obligated to terminate and result in a state satisfying the postcondition.

Above, we discuss the derivation of a procedure's implied postcondition in terms of its given precondition and state that the derived information can be used to evaluate the procedure's correctness with respect to its contracted obligation. But what if the caller of the procedure has not met its obligation? In this section we present a technique for determining if a procedure call is made to a procedure under the required conditions. That is, for any procedure, `proc`, this technique will establish if the calling context ensures the precondition of `proc`, Q_{proc}, prior to calling it.

We wish to determine if the actual parameters to the procedure are in a state that satisfies the constraints on the associated formal parameters. Let B describe the state prior to a procedure call `proc(`al`)` to a procedure `proc(`pl`)`. Substituting actual for formal parameters in the procedure's initial conditions Q_{proc} allows us to investigate whether the actual parameters satisfy the constraints on the formal parameters. This gives the three new substitution rules.

- For value parameters:

$$Q'_{proc} \equiv Q_{proc}\left[al_i/pl_i?\right] \tag{8.9}$$

- For result parameters:

$$Q'_{proc} \equiv Q_{proc}\left[al_i/pl_i\right] \tag{8.10}$$

- For reference parameters:

$$Q'_{proc} \equiv Q_{proc}\left[e, al_i/pl_i?, pl_i\right] \tag{8.11}$$

  where $B \implies al_i = e$

If it is not the case that $B \Rightarrow Q'_{proc}$, then the values of the actual parameters immediately prior to the procedure call do not satisfy the constraints imposed by the called procedure's precondition. If $\neg\left(B \Rightarrow Q'_{proc}\right)$, that is, $\left(B \wedge \neg Q'_{proc}\right)$, is satisfiable, then the client of the procedure may not satisfy its contracted obligation. If this is the case, we can describe the required obligations by Q'_{proc}. The approach to checking the precondition is summarised in Algorithm 13.

We define the following inspection rule to identify precondition satisfiability problems with procedure calls:

Inspection Rule 5 (Satisfiable Procedure Precondition). For a procedure call to `proc`, apply Alorithm 13. Determine if $B_i \wedge \neg Q'_{proc}$ is satisfiable for every disjunct, B_i, of the initial conditions, B, immediately prior to the procedure call.

If $B_i \wedge \neg Q'_{proc}$ is satisfiable, then the precondition for the procedure `proc` will not be satsified in any state that satisfies $B_i \wedge \neg Q'_{proc}$. If $B_i \wedge \neg Q'_{proc}$ is not satisfiable, then the precondition for the procedure `proc` will always be satsified under the initial conditions B_i.

Algorithm 13 Checking a procedures precondition

Consider the procedure call `proc(`al`)` to a procedure `proc(`pl`)`, made under the conditions B, where al is a parameter list of actual parameters positionally associated with the formal parameter list pl and B is a formula of the form $B_1 \vee ... \vee B_n$ where the sub formulae $B_1..B_n$ contain no disjuncts.
For all $i \in [1..n]$, A procedure call may fail, due to not meeting the called procedure's precondition, under the conditions calculated as follows:
For each B_i

1. Calculate Q'_{proc} by applying (8.9), (8.10) and (8.11) with Q_{proc} and B_i.

2. Evaluate
$$B_i \wedge \neg Q'_{proc}$$

A reviewer may address the reasoning in this inspection rule either by using a theorem prover or by manual, human reasoning.

Applying Algorithm 13 to the call to `add` in the example above, we see that the call is made under the initial conditions

$$B = (a = a? \wedge b = b? \wedge a, b, s \in \texttt{seq}\ \mathbb{Z})$$

We have previously calculated Q_{add} by (8.3) as

$$Q_{add} \equiv (\#v = \#w \wedge v = v? \wedge v \in \texttt{seq}\ \mathbb{Z} \wedge w = w? \wedge w \in \texttt{seq}\ \mathbb{Z} \wedge sum \in \texttt{seq}\ \mathbb{Z})$$

Refining Q_{add} by applying (8.9), (8.10) and (8.11), gives

$$\begin{array}{rcl} Q'_{add} & \equiv & (\#v = \#w \wedge v = a \wedge v \in \texttt{seq}\ \mathbb{Z} \wedge w = b \wedge w \in \texttt{seq}\ \mathbb{Z} \wedge s \in \texttt{seq}\ \mathbb{Z}) \\ & \equiv & (\#a = \#b \wedge a \in \texttt{seq}\ \mathbb{Z} \wedge b \in \texttt{seq}\ \mathbb{Z} \wedge s \in \texttt{seq}\ \mathbb{Z}) \end{array}$$

We then evaluate

$$\begin{array}{ll} & B \wedge \neg Q'_{add} \\ \equiv & \begin{array}{l} a = a? \wedge b = b? \wedge a, b, s \in \texttt{seq}\ \mathbb{Z} \wedge \\ (\neg(\#a = \#b) \vee a \notin \texttt{seq}\ \mathbb{Z} \vee b \notin \texttt{seq}\ \mathbb{Z} \vee s \notin \texttt{seq}\ \mathbb{Z}) \end{array} \\ \equiv & \begin{array}{l} (a = a? \wedge b = b? \wedge a, b, s \in \texttt{seq}\ \mathbb{Z} \wedge \#a \neq \#b) \vee \\ (a = a? \wedge b = b? \wedge a, b, s \in \texttt{seq}\ \mathbb{Z} \wedge a \notin \texttt{seq}\ \mathbb{Z}) \vee \\ (a = a? \wedge b = b? \wedge a, b, s \in \texttt{seq}\ \mathbb{Z} \wedge b \notin \texttt{seq}\ \mathbb{Z}) \vee \\ (a = a? \wedge b = b? \wedge a, b, s \in \texttt{seq}\ \mathbb{Z} \wedge s \notin \texttt{seq}\ \mathbb{Z}) \end{array} \\ \equiv & (a = a? \wedge b = b? \wedge a, b, s \in \texttt{seq}\ \mathbb{Z} \wedge \#a \neq \#b) \end{array}$$

By the satisfiable procedure precondition inspection rule, we reason that $B \wedge \neg Q'_{add}$ is satisfiable whenever $a = a? \wedge b = b? \wedge a, b, s \in \texttt{seq}\ \mathbb{Z} \wedge \#\mathbf{a} \neq \#\mathbf{b}$. So, whenever $\#a \neq \#b$ prior to calling `add`, the precondition for `add` will not be satisfied.

8.4.3 Strongest Postcondition for Procedure Call

Based on the properties of substitution and the parameter mechanisms defined above, we are now in a position to define an algorithm for calculating the strongest postcondition of a procedure call. Algorithm 14 defines a process for calculating the strongest postcondition for procedure calls based on substitution into an abstract form of the called procedure's specification. It includes precondition checking.

Algorithm 14 *sp* for procedure call

Consider the procedure call `proc(`al`)` to a procedure `proc(`pl`)`, under some precondition Q, where al is a parameter list of actual parameters positionally associated with the formal parameter list pl and Q is a formula of the form $Q_1 \vee ... \vee Q_n$ where the sub formulae $Q_1..Q_n$ contain no disjuncts.
$sp(Q, proc(al)) = sp\,(Q_1, proc(al)) \vee .. \vee sp\,(Q_n, proc(al))$, where for all $i \in [1..n]$, $sp\,(Q_i, proc(al)) \equiv Q'_i$ where Q'_i is defined as:

1. Apply algorithm 13 to determine if the precondition for `proc` is satisfied immediately prior to the call. If this algorithm does not yield $false$, then warn that the following procedure-call semantics are only defined for states in Q'_{proc}.
2. Calculate R'_{proc} by applying (8.6), (8.7) and (8.8).
3. Simplify R'_{proc} by applying (8.5), to remove the references to the formal parameters.
4. For all $i \in [1..n]$, calculate Q'_i by the following[4]:
   (a) For every predicate in Q_i involving a variable, v, in the actual parameter list which is positionally associated with a formal *reference* or *result* parameter, where $Q_i \implies v = E$ and E is an expression, substitute E for v in Q_i (if there is no explicit E use v_0. This is to ensure that we do not lose any information as a result of v being changed by the procedure. Note, that if the procedure does not actually change v then $v = E$ is maintained in R'_{proc} after step 2.
   (b) Remove every predicate in Q_i involving a variable, v, in the actual parameter list which is positionally associated with a formal *result* parameter. This is necessary, as the procedure is responsible for defining the value of a result parameter.
   (c) Conjoin R'_{proc} from step 2 to Q_i to give Q'_i.

$$Q'_i \equiv Q_i \wedge R'_{proc}$$

Consider again the following call to `add`.

$$\{a = a? \wedge b = b? \wedge a, b, s \in \texttt{seq}\ \mathbb{Z}\}$$

```
add(a,b,s);
```

We apply algorithm 14, to compute the postcondition of this procedure call, $sp\,(Q,\ \texttt{add}(\texttt{a},\texttt{b},\texttt{s}))$, where

$$Q \equiv (a = a? \wedge b = b? \wedge a, b, s \in \texttt{seq}\,\mathbb{Z})$$

Step 1: We checked that Q ensures the procedure precondition Q'_{add} by Algorithm 13 in the previous section and reasoned that the procedure precondition is satisfied, and therefore the calculated postcondition is only guaranteed, when $\#a = \#b$ prior to the call.

Step 2: R'_{add} was calculated in the previous section to be:

$$R'_{add} \quad \equiv \quad \left(\begin{array}{l} \#a = \#b \wedge a \in \texttt{seq}\,\mathbb{Z} \wedge b \in \texttt{seq}\,\mathbb{Z} \wedge \\ s \in \texttt{seq}\,\mathbb{Z} \wedge \forall j \in [0..\#v-1]\,(s[j] = a[j] + b[j]) \end{array} \right)$$

Step 3: There are no references to formal parameters in R'_{add} so no substitution is required.

Step 4: We remove all of the references to s in the procedure call precondtion Q giving $Q \equiv (a = a? \wedge b = b? \wedge a, b, s \in \texttt{seq}\,\mathbb{Z})$.

Conjoining R'_{add} to $(a = a? \wedge b = b? \wedge a, b, s \in \texttt{seq}\,\mathbb{Z})$ gives:

$$Q' \equiv \left(\begin{array}{l} \#a = \#b \wedge a, b, s \in \texttt{seq}\,\mathbb{Z} \wedge \\ \forall j \in [0..\#v-1]\,(s[j] = a[j] + b[j]) \wedge \\ a = a? \wedge b = b? \end{array} \right)$$

There is only one disjunct in Q, so

$$\begin{array}{ll} & sp(Q, \texttt{add}(\texttt{a}, \texttt{b}, \texttt{s})) \\ \equiv & \left(\begin{array}{l} \#a = \#b \wedge a, b, s \in \texttt{seq}\,\mathbb{Z} \wedge \\ \forall j \in [0..\#v-1]\,(s[j] = a[j] + b[j]) \wedge \\ a = a? \wedge b = b? \end{array} \right) \end{array}$$

when $\#a = \#b$.

8.5 Functions

Although functions can be dealt with as a procedure with `OUT` parameters representing the returns, we present a discussion of function and function call semantics separately from procedures and procedure calls. Functions are special procedures which return values. In our example language functions are declared with the following syntax, where `fn` is the name of a function, *pl* is a formal parameter list and (T) is the type of the value returned:

```
function fn(pl) :  T
```

A function is a block that represents an abstraction of an expression. Functions may be used as any other expression is used. This carries the implication that functions may be used on the right hand side of an assignment, such as:

```
top := pop(stack);
```

Functions may also be used as arguments to comparison operators, such as:

```
if empty(stack) = False ⟶ ...
```

To determine the semantics of a function and function call, we again define an approach to deriving the semantics of the function abstraction itself and to using this derived abstraction for defining a function call.

Deriving the semantics of the function abstraction is very similar to deriving the semantics of a procedure abstraction. The only difference is that we need to represent the return value. In our language the value to be returned from a function is specified by assigning it to a variable with the same name as the function itself, as is done in languages like Pascal. For example, if (exp) is an expression, then the following returns that expression from the function `fn`:

```
fn := exp;
```

As such, the return has the same semantics as an assignment to a special `OUT` parameter with the name of the function[5]. The assignment yields the equality $fn = exp$. In addition to this we know from the function declaration that this special variable is of type, `T`. To summarise, for any function `fn`, returning a value of type `T`, a return of the form `fn:=exp`, yields $fn = exp \wedge fn \in T$.

As an example of deriving the semantics of a function abstraction, consider the following function to return the top element of a non-empty stack:

$\{\#s > 0\}$
```
function top( IN s:  seq  T) : T {
    top:=s[#s-1];
}
```

The semantics for the function abstraction, as calculated by algorithm 12, is

$$R_{top} \equiv (\#s > 0 \wedge s = s? \wedge s \in \texttt{seq}\, T \wedge top = s[\#s - 1] \wedge top \in T)$$

As mentioned previously, functions can be used as expressions in assignments and as operands in boolean expressions. For either use, the correct semantics

[5] Although we have decided to use an assignment notation for returns, the same semantics are involved with other common function return notations, such as the `return` keyword in Java or C like languages

can be defined by applying algorithm 15. This algorithm differs from that for calculating procedure call semantics in that this one makes substitutions in the expression containing the function call not the precondition. As such it is a substitution algorithm, not an instance of a predicate transformer. By applying algorithm 15 to substitute in the expression containing the procedure call, the existing strongest postcondition predicate transformers may be applied to the resulting expression and statement in which that expression is used.

Consider the following example:

$\{stack \in \texttt{seq}\, T \wedge t \in T\}$

```
if !empty(stack) ⟶

    t:=top(stack);

[] empty(stack) ⟶

    // Do Nothing

fi
```

given the following function specifications:

$\{true\}$

```
function empty(IN s :  seq T) : 𝔹 { ...  }
```

$\{s = s? \wedge s \in \texttt{seq}\, T \wedge empty = (\#s = 0)\}$

$\{\#s > 0\}$

```
function top(IN s :  seq T) : T { ...  }
```

$\{s = s? \wedge s \in \texttt{seq}\, T \wedge top = s[\#s - 1]\}$

Then, applying algorithm 15:

Step 1: Check the precondition is satisfied

The precondition for `empty` is the weakest possible precondition $true$. It is always satisfied (applying Algorithm 13 yields $false$).

Step 2: Calculate R'_{empty}

Applying (8.6), (8.7) and (8.8), R'_{empty} is calculated as:

$$R'_{empty} \equiv (s = stack \wedge s \in \texttt{seq}\, T \wedge empty = (\#s = 0))$$

Step 2: Remove references to formal parameters by substitution

Applying (8.5), R'_{empty} is refined to:

$$R'_{empty} \equiv (stack \in \texttt{seq}\, T \wedge empty = (\#stack = 0))$$

Step 4: Determine semantic effects of function call

Algorithm 15 *sp* for function call

Consider the function call `fn(`al`)` to a function `fn(`pl`):T`, used in an expression as part of a statement, under some precondition Q, where Q is a formula of the form $Q_1 \vee ... \vee Q_n$ where the sub formulae $Q_1..Q_n$ contain no disjuncts.
Let *stmt* be the statement involving the function call, then the strongest postcondition of *stmt* under Q is given by the following calculations, where the function `fn` is accessible and R_{fn} has been calculated:

1. Apply Algorithm 13 to determine if the precondition for `fn` is satisfied immediately prior to the call. If this algorithm does not yield $false$, then warn that the following procedure-call semantics are only defined for states in Q'_{fn}.
2. Calculate R'_{fn} by applying (8.6), (8.7) and (8.8).
3. Simplify R'_{fn} by applying (8.5), to remove the references to the formal parameters.
4. For all $i \in [1..n]$, calculate Q'_i by the following[6]:
   (a) For every predicate in Q_i involving a variable, v, in the actual parameter list which is positionally associated with a formal *reference* or *result* parameter, where $Q_i \implies v = E$ and E is an expression, substitute E for v in Q_i (if there is no explicit E use v_0). This is to ensure that we do not lose any information as a result of v being changed by the procedure. Note, that if the procedure does not actually change v then $v = E$ is maintained in R'_{proc} after step 2.
   (b) Remove every predicate in Q_i involving a variable, v, in the actual parameter list which is positionally associated with a formal *result* parameter. This is necessary, as the procedure is responsible for defining the value of a result parameter.
   (c) Conjoin R'_{fn} from step 2 to Q_i to give Q'_i.
   $$Q'_i \equiv Q_i \wedge R'_{fn}$$
5. For each Q'_i calculate the respective strongest postcondition R_i of *stmt* under Q'_i. That is, for every Q_i:
   $$R_i \equiv sp\left(Q'_i, stmt\right)$$
6. As every function must return a value to be syntactically correct, then R_{fn} will contain a conjunct of the form $fn = exp$. By the above calculations, this equality must now be a conjunct of each R_i. We apply the following substitution to remove the function call in the postcondition:
   $$R_i \equiv R_i\,[\texttt{exp}/\texttt{fn}(al)]\,[\texttt{exp}/fn]$$

There is only one disjunct in the given precondition, so we deal only with Q. There are no reference or result parameters in the parameter list of `empty`. Therefore, the precondition, Q, prior to the function call remains unchanged as $(stack \in \texttt{seq}\, T \wedge t \in T)$, and Q' is given by

$$\begin{array}{lll} Q' & \equiv & R'_{empty} \wedge Q \\ & \equiv & (stack \in \texttt{seq}\, T \wedge empty = (\#stack = 0)) \wedge (stack \in \texttt{seq}\, T \wedge t \in T) \\ & \equiv & (stack \in \texttt{seq}\, T \wedge empty = (\#stack = 0) \wedge t \in T) \end{array}$$

Step 5: Calculated the strongest postcondition of `if !empty(stack)`

There is only one disjunct in Q', so we calculate the postcondition of `if empty(stack)`, denoted R by

$$\begin{array}{lll} R & \equiv & sp\,(Q',\ \texttt{if !empty(stack)}\ ) \\ & \equiv & sp\,((stack \in \texttt{seq}\, T \wedge empty = (\#stack = 0) \wedge t \in T)\,,\ \texttt{if !empty(stack)}\ ) \\ & \equiv & (stack \in \texttt{seq}\, T \wedge empty = (\#stack = 0) \wedge t \in T \wedge \neg empty(stack)) \end{array}$$

This result may be useful for verification and should be noted. In order to completely abstract the semantics of the function, we apply the following step.

Step 6: Abstract out the function call

R implies the equality, $empty = (\#stack = 0)$. This equality can be used to replace any references to the function call to `empty()` in R by the abstracted semantics for the call to `empty()`, giving

$$\begin{array}{lll} R & \equiv & \left(\begin{array}{l} stack \in \texttt{seq}\, T \wedge t \in T \wedge \\ empty = (\#stack = 0) \wedge \neg empty(stack) \end{array} \right) \begin{array}{l} [(\#stack = 0)/empty(stack)] \\ [(\#stack = 0)/empty] \end{array} \\ & \equiv & (stack \in \texttt{seq}\, T \wedge t \in T \wedge \neg(\#stack = 0)) \\ & \equiv & (stack \in \texttt{seq}\, T \wedge t \in T \wedge (\#stack \neq 0)) \end{array}$$

■

So, the strongest postcondition of the statement in the first branch is given by the following, where the semantics of the call to `top(stack)` are calculated in a similar manner to those for `empty()`:

$$\begin{array}{ll} & sp\,((stack \in \texttt{seq}\, T \wedge t \in T \wedge (\#stack \neq 0))\,,\ \texttt{t} := \texttt{top(stack)}) \\ \equiv & \left(\begin{array}{l} stack \in \texttt{seq}\, T \wedge t \in T \wedge \\ (\#stack > 0) \wedge t = top(stack) \end{array} \right) \begin{array}{l} [(stack[\#stack - 1]/top(stack)] \\ [(stack[\#stack - 1]/top] \end{array} \\ \equiv & (stack \in \texttt{seq}\, T \wedge t \in T \wedge (\#stack > 0) \wedge t = stack[\#stack - 1]) \end{array}$$

Note that the precondition of `top` is always established prior to the call to `top` in this branch, as a result of the branch guard.

Similarly the strongest postcondition of the second branch is calculated as:

$$(stack \in \texttt{seq}\, T \wedge t \in T \wedge (\#stack = 0))$$

And the postcondition for the whole block is

$$\begin{array}{l} (stack \in \texttt{seq}\, T \wedge t \in T \wedge (\#stack > 0) \wedge t = stack[\#stack - 1]) \vee \\ (stack \in \texttt{seq}\, T \wedge t \in T \wedge (\#stack = 0)) \end{array}$$

As we are concerned with the readability of these specifications for the purpose of inspection, the derived postcondition should be used in the above form as well as the form established prior to the last step of algorithm 15.

For example, for the above, the following specification form might be more useful for reasoning if the semantics of `empty()` and `top()` are assumed to be understood:

$$(\mathit{stack} \in \mathtt{seq}\, T \wedge t \in T \wedge \neg \mathit{empty}(\mathit{stack}) \wedge t = \mathit{top}(\mathit{stack})) \vee$$
$$(\mathit{stack} \in \mathtt{seq}\, T \wedge t \in T \wedge \mathit{empty}(\mathit{stack}))$$

Any annotation of the code with specifications for the purpose of assisting reasoning should present both forms.

8.6 Related Work and Conclusion

Gries [47] presents a theorem for specifying the semantic effects of executing a procedure call. This theorem provides a means by which the specification of a procedure call can be constructed by in-lining the body of the called procedure and multiple assignments representing parameter instantiation and returns in the context of the procedure call [47, 38]. This theorem was paraphrased in section 3.2.5. Gannod and Cheng [38] use this formulation to derive specifications from programs to support reverse engineering. They inline a procedure call, achieve parameter binding by multiple assignment, and establish a postcondition by applying the *sp* for assignment to the in-lined procedure body (so long as the procedure is non-recursive and contains no iteration[38]).

In this chapter we have presented an approach for deriving specifications from procedures and functions as abstract units of code. In these specifications, formal parameters are initially bound to values which represents corresponding actual parameters. That is, for a formal parameter v the equality $v = v?$ is established, where $v?$ represents the actual parameter. This abstracted specification can then be stored allowing the postcondition of a call to that procedure to be calculated by substitution into the stored specification. This technique provides a mechanism by which procedure precondition violations and repeated formal reference parameter defects can be identified. The main advantage of this approach over [47] and [38] is that calculation of procedure and function call semantics is reduced to substitution in this approach. Gries's theorem implies that whenever a call to a procedure is made the strongest postcondition of the procedure body needs to be calculated. Both approaches provide a mechanism by which local variables are removed from the specification by substitution, ensuring that the derived postconditions for procedure and function calls contain no references to unnecessary local (to the procedure) variables.

Finally, the approach taken in this chapter provides a means by which we can interactively yield a postcondition for calls to procedures or functions for which specifications are not automatically derivable. For example, consider a procedure or function for which an automated tool can not yield a strong postcondition, implying that there is no stored postcondition for the procedure or function. A possible strategy to handle such cases is to suggest a strong postcondition for the procedure or function, verify with the aid of the derived information that the suggested postcondition is satisfied by the procedure or function (see next chapter), and store the suggested postcondition for use in calculating procedure call semantics by substitution. The approach of inlining a procedure can not support such a strategy and is, therefore, limited in its applicability.

Chapter 9

Verification

In chapter 1 the concepts of functional correctness and verification were introduced. This chapter describes verification based on the use of derived semantics. It describes techniques for using the derived semantic information in constructing formal correctness proofs.

Here we discuss an approach to use a derived specification to support verification using both the axiomatic approach, as described by Floyd [34] and Hoare [53], and the function-theoretic approach as described in Cleanroom software engineering [67]. The choice of approach taken is dependent on the type of specification given for the program being verified and the level of specification that could be derived from the program.

If a specification is given in first-order predicate logic, then we apply axiomatic verification techniques. That is, given a precondition Q and postcondition R in predicate logic, for a program S, we can use the specifications derived through application of the methods of the previous chapters to show that $\{Q\}S\{R\}$ and to reason that S terminates normally. When S contains no iterative constructs such a proof is trivial. It is not always possible to directly prove $\{Q\}S\{R\}$, when S contains iterative structures. In terms of the derivation techniques presented in earlier chapters, if the strongest accessible conjunctive invariant is calculable, then direct verification of $\{Q\}S\{R\}$ is possible, otherwise it is not.

Hoare's inference rules for loop verification simplify the problem of verifying loops by requiring the identification of a loop invariant P. Verification then involves demonstrating that the loop maintains the invariant P and that P and the negation of the loop guard implies the required loop postcondition.

This chapter presents techniques for both the direct method and a method based on mathematical induction for proving the maintenance of loop invariants. These methods use only derived information and the given specification.

If a specification is given as a Cleanroom box specification defining the intended function of a program or program segment, then we can apply the Cleanroom function-theoretic verification approach. This chapter presents a method for supporting this approach using only derived information and the given specification.

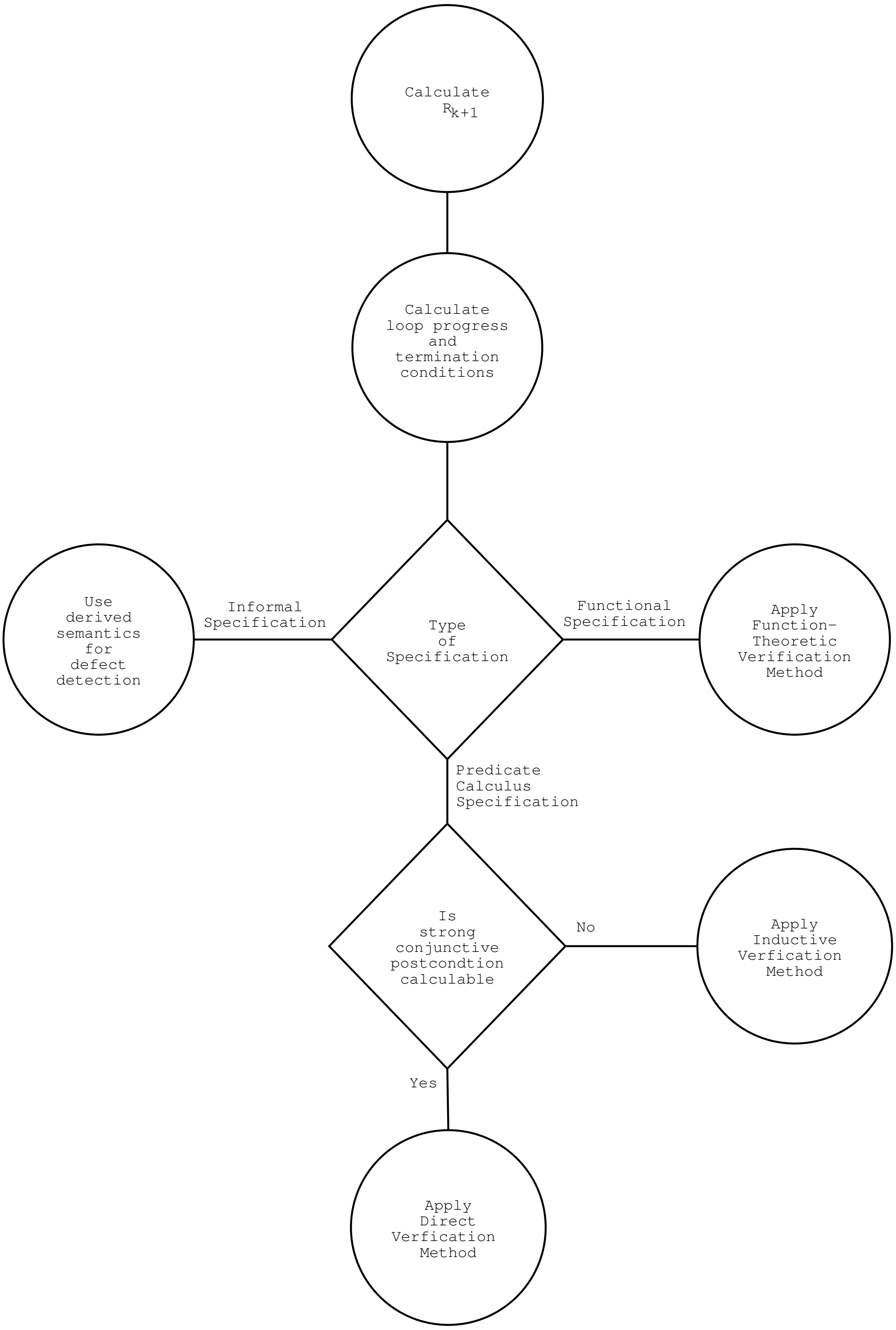


Figure 9.1: Approaches to Loop Verification

Figure 9.1 presents diagrammatically the three approaches to formal verification discussed in this chapter. Also illustrated are the conditions for performing informal defect detection rather than verification. The methods discussed in this chapter concern only proof of partial correctness. Verification questions and techniques for investigating termination were presented previously in chapter 7.

9.1 Axiomatic Verification Method

As illustrated in Fig. 9.1, we present two approaches for axiomatic verification of loops. The choice of approach is dependent on whether a strong, conjunctive invariant, and hence, postcondition is derivable from the loop. If such a postcondition is derivable, then the loop can be verified directly. The direct approach is discussed at the end of this section.

An alternate approach, which is applicable even when a strong conjunctive postcondition is not derivable, is based on Hoare's inference rules for iterative structures [53] and the corresponding fundamental invariance theorem [22]. This approach is based on a proof by mathematical induction that a loop maintains a given invariant, P, and that the conjunction of P and the negation of the loop guard establishes the required postcondition for that loop. Providing a suitable invariant, P is the main restriction on this method. By definition [26], a loop invariant must hold before and after every iteration of the loop for which it is invariant. If we are given a postcondition for a loop, say R, it may be possible to weaken that postcondition to describe a loop invariant, P, such that $P \wedge \neg G \Rightarrow R$, where G is the loop guard. Such an invariant will be referred to as a *suitable invariant.*

The following inference rule from Hoare [53] states that if a suitable invariant P is available for a loop with a postcondition R, and it can be shown that P holds before and after execution of the loop body (the guard must be true), then it can be inferred that the loop is partially correct with respect to R.

$$\frac{\{P \wedge G\}S\{P\}}{\{P\}\mathtt{do}\ \ G \rightarrow \ \ S\ \ \mathtt{od}\{P \wedge \neg G\}}$$

We state the fundamental invariance theorem.

Theorem 9.1 (Fundamental Invariance Theorem). For any loop, `do G` $\rightarrow$ `S od`, if P is an invariant of that loop then,

- $\{P \wedge G\}$ `S` $\{P\}$ holds. If the invariant, P holds and the guard G holds prior to execution of S, then S terminates satisfying P. A proof that this occurs is a proof of partial correctness.
- If `do G` $\rightarrow$ `S od` terminates, then $P \wedge \neg G$ will hold on termination, and $(P \wedge \neg G) \Rightarrow R$, where R is the loop postcondition.

It follows from the fundamental invariance theorem, that in order to verify $\{Q\}$ `do G` $\rightarrow$ `S od` $\{R\}$, we must identify a suitable invariant P such that $P \wedge \neg G \Rightarrow R$, and then prove the following:

- P is true immediately prior to execution of the loop (immediately after initialisation). That is, we must prove

$$Q \Rightarrow P \tag{9.1}$$

- P is invariant. That is, we must prove

$$\{P \wedge G\}\mathtt{S}\{P\} \tag{9.2}$$

Many techniques including [26, 24] use the weakest precondition predicate transformer to prove that a loop with loop body S and guard G maintains a given invariant P by showing that

$$(P \wedge G) \Rightarrow wp\,(\mathtt{S},\ P) \tag{9.3}$$

This technique is disadvantaged by the fact that automating the application of the weakest precondition predicate transformer is non-trivial due to the need to calculate inverse functions.

Alternatively, we may employ the strongest postcondition predicate transformer to prove that a loop with loop body S and guard G satisfies (9.2) by proving the following:

$$sp\,(P \wedge G,\ \mathtt{S}) \Rightarrow P \tag{9.4}$$

As described in chapter 3 of this thesis, we have been able to automate application of the sp predicate transformer for non-iterative constructs. So $sp\,(P \wedge G,\ \mathtt{S})$ is calculable, when initial values for all variables accessible to `S` are known. These values are often difficult to find.

Consider, for example, an inductive proof that the loop in Program 14 maintains the invariant $P \equiv (ans \times b^e = base^{exp} \wedge e \geq 0)$.

Proof. The proof is by induction:

Base Case

Show $Q \Rightarrow P$:

$$\begin{array}{ll} & (ans = 1 \wedge b = base \wedge e = exp) \Rightarrow (ans \times b^e = base^{exp} \wedge e \geq 0) \\ \equiv & true \end{array}$$

Program 14 Base to power example loop

```
Q ≡ {ans = 1 ∧ b = base ∧ e = exp}
do e > 0 →
    if e mod 2 ≠ 0 →
        ans := ans*b;
        e := e - 1;
    [] e mod 2 = 0 →
        b := b*b;
        e := e div 2;
    fi;
od
```

Inductive Step

Assume P holds after some arbitrary kth iteration. Then show it holds for the $k+1$st iteration.

By (9.4):

$$\begin{aligned} & sp\,(ans \times b^e = base^{exp} \wedge e \geq 0 \wedge G,\ \mathtt{S}) \Rightarrow (ans \times b^e = base^{exp} \wedge e \geq 0) \\ \equiv\ & sp\,(ans \times b^e = base^{exp} \wedge e > 0,\ \mathtt{S}) \Rightarrow (ans \times b^e = base^{exp} \wedge e \geq 0) \\ \equiv\ & \ldots \end{aligned}$$

The sp above is difficult to calculate (and particularly automate) as a result of the necessity to calculate from P initial values for ans, b and e. □

As is the case above, the strongest postcondition calculation in (9.4) is often non-trivial as P may not describe initial states for all program variables, may define the initial state of a variable in terms of a function that is difficult to invert, or may involve quantified formulae. One way to simplify the proof is to introduce arbitrary values for the variables on commencement of some arbitrary $k+1$st iteration and calculate the changes on that iteration. For this purpose we use the, already calculated, iterative-form invariant, R_{k+1}. The calculation of R_{k+1} is able to be automated as Q_{k+1}, the initial state from which R_{k+1} is calculated, defines arbitrary values for all accessible variables, so there is no need to calculate inverses of functions, and there are no quantified formulae in Q_{k+1}. The next section investigates the use of R_{k+1} for proving that (9.2) is maintained for a given loop and invariant.

9.1.1 Proving Partial Correctness using R_{k+1}

For any given loop invariant, P, P_k represents the fact that P holds after a kth iteration ($k \geq 0$). This is a use of the notation presented in chapter 3, so

P_k is obtained by subscripting all non-constant free variables in P with k in accordance with the following:

$$P_k = P\left[v_k/v\right], \text{ for all free variables } v \text{ accessible to the loop} \tag{9.5}$$

P_k, therefore, describes an invariant state that must hold immediately after an arbitrary kth iteration of the loop.

For example, given an invariant $P = \{i \leq N\}$, $P_k \equiv \{i_k \leq N\}$.

Similarly, P_{k+1} defines the invariant P in terms of $k+1$st iteration variable instances. That is, P_{k+1} describes an invariant state that must hold at the end of a $k+1$st iteration of a loop.

If an invariant, P, holds at the beginning of some arbitrary $k+1$st iteration ($k \geq 0$), then P_k must hold. Because the loop must iterate for a $k+1$st time, then G must also hold after the kth iteration. That is, G_k holds. At the beginning of a $k+1$st iteration, every accessible variable, v, will hold some value equivalent to v_k. That is, $v_{k+1} = v_k$ holds initially for any variable v on a $k+1$st iteration. The initial conditions on an arbitrary $k+1$st iteration, Q_{k+1}, defined in chapter 4 asserts G_k and $v_{k+1} = v_k$ for all accessible variables v. Therefore, in order to prove that a loop, Q `do G` $\rightarrow$ `S od` maintains the invariant P, by the fundamental invariance theorem, we prove (9.1) and the following:

$$sp\left(P_k \wedge Q_{k+1}, S\right) \Rightarrow P_{k+1} \tag{9.6}$$

The strongest postcondition in this form is calculable as Q_{k+1} provides a definition for every variable at the commencement of the $k+1$st iteration of the loop. We may, however, simplify it further by recognising that we have already applied the *sp* calculation in computing R_{k+1}.

The invariant at the commencement of a $k+1$st iteration, P_k refers only to kth iteration instances of the loop variables. Any assignment statements on the $k+1$st iteration affect only $k+1$st iteration instances of the loop variables. It follows that, any terms involving only kth iteration instances of the loop variables are unchanged by the loop body assignments and are, therefore, implied in the strongest postcondition. So, if P_k holds before commencement of a $k+1$st iteration it must hold after that iteration. We can, therefore, simplify (9.6) as follows:

$$\begin{aligned} & sp\left(P_k \wedge Q_{k+1}, S\right) \Rightarrow P_{k+1} \\ \equiv\ & P_k \wedge sp\left(Q_{k+1}, S\right) \Rightarrow P_{k+1} \\ \equiv\ & P_k \wedge R_{k+1} \Rightarrow P_{k+1} \ \ldots \ \texttt{by definition of } R_{k+1} \end{aligned} \tag{9.7}$$

To summarise, we have the following theorem.

Theorem 9.2. [Iterative Inductive Theorem]

For any loop, $\{Q\}$ `do` $G \rightarrow S$ `od`, to prove partial correctness with respect to a given invariant P, apply the following inductive proof steps:

1. **Base Case:**

   Prove that the given invariant P is maintained on loop initialisation by showing (9.1):
   $$Q \Rightarrow P$$

2. **Inductive Step:**

   Prove by (9.7) that when the invariant holds at the start of an arbitrary $k+1$st iteration $k \geq 0$ and the loop guard is true at the start of this iteration, then the invariant is maintained after the iteration:
   $$P_k \wedge R_{k+1} \Rightarrow P_{k+1}$$

Consider again the *power* example in Program 14. By theorem 9.2 the inductive proof of partial correctness is as follows:

Base Case:

Show $Q \Rightarrow P$:

$$\begin{array}{ll} & (ans = 1 \wedge b = base \wedge e = exp) \Rightarrow (ans \times b^e = base^{exp} \wedge e \geq 0) \\ \equiv & true \end{array}$$

Inductive Step:

Given that P holds after a kth iteration ($k \geq 0$), that is given P_k, show that P_{k+1} holds immediately after the $k+1$st iteration.

By (9.7) we show that $P_k \wedge R_{k+1} \Rightarrow P_{k+1}$.

P_k is given as:
$$ans_k \times {b_k}^{e_k} = {base_k}^{exp_k} \wedge e_k \geq 0$$

So, P_{k+1} is:
$$ans_{k+1} \times {b_{k+1}}^{e_{k+1}} = {base_{k+1}}^{exp_{k+1}} \wedge e_{k+1} \geq 0$$

We calculate R_{k+1} by the procedure presented in chapter 4 as

$$\left(\begin{array}{l} base_{k+1} = base_k \wedge exp_{k+1} = exp_k \wedge e_k > 0 \wedge \\ \left(\begin{array}{l} (e_k mod 2 \neq 0 \wedge ans_{k+1} = ans_k \times b_k \wedge b_{k+1} = b_k \wedge e_{k+1} = e_k - 1) \vee \\ (e_k mod 2 = 0 \wedge ans_{k+1} = ans_k \wedge b_{k+1} = {b_k}^2 \wedge e_{k+1} = \frac{e_k}{2}) \end{array} \right) \end{array} \right)$$

Then,

$$\begin{array}{ll} & P_k \wedge R_{k+1} \\ \equiv & \left(\begin{array}{l} ans_k \times {b_k}^{e_k} = {base_k}^{exp_k} \wedge \\ base_{k+1} = base_k \wedge exp_{k+1} = exp_k \wedge e_k > 0 \wedge \\ \left(\begin{array}{l} (e_k mod 2 \neq 0 \wedge ans_{k+1} = ans_k \times b_k \wedge b_{k+1} = b_k \wedge e_{k+1} = e_k - 1) \vee \\ (e_k mod 2 = 0 \wedge ans_{k+1} = ans_k \wedge b_{k+1} = {b_k}^2 \wedge e_{k+1} = \frac{e_k}{2}) \end{array} \right) \end{array} \right) \end{array}$$

We prove that $P_k \wedge R_{k+1} \Rightarrow P_{k+1}$. This proof obligation is generated by the prototype tool whenever an invariant is provided for a loop. The proof obligation for this example can be seen in the tool output of Fig. **??** in Appendix B.

$P_k \wedge R_{k+1}$ describes two cases, corresponding to the two guarded paths through the loop body. Both paths must maintain the invariant[1], so they are treated independently. We prove the following two cases

$$\begin{array}{lcl} P_k \wedge R_{1,k+1} & \Rightarrow & (ans_{k+1} \times {b_{k+1}}^{e_{k+1}} = {base_{k+1}}^{exp_{k+1}} \wedge e_{k+1} \geq 0) \\ P_k \wedge R_{2,k+1} & \Rightarrow & (ans_{k+1} \times {b_{k+1}}^{e_{k+1}} = {base_{k+1}}^{exp_{k+1}} \wedge e_{k+1} \geq 0) \end{array}$$

Case1:

Prove

$$P_k \wedge R_{1,k+1} \Rightarrow (ans_{k+1} \times {b_{k+1}}^{e_{k+1}} = {base_{k+1}}^{exp_{k+1}} \wedge e_{k+1} \geq 0)$$

That is, given

$$\left(\begin{array}{l} ans_k \times {b_k}^{e_k} = {base_k}^{exp_k} \wedge \\ base_{k+1} = base_k \wedge exp_{k+1} = exp_k \wedge e_k > 0 \wedge \\ e_k mod 2 \neq 0 \wedge ans_{k+1} = ans_k \times b_k \wedge b_{k+1} = b_k \wedge e_{k+1} = e_k - 1 \end{array}\right)$$

we show that we can infer

$$ans_{k+1} \times {b_{k+1}}^{e_{k+1}} = {base_{k+1}}^{exp_{k+1}} \wedge e_{k+1} \geq 0$$

The proof is by contradiction; that is, given $P_k \wedge R_{1,k+1}$, show that supposing $ans_{k+1} \times {b_{k+1}}^{e_{k+1}} \neq {base_{k+1}}^{exp_{k+1}} \vee e_{k+1} < 0$ results in a contradiction.

1.	$base_{k+1} = base_k$	(given)
2.	$exp_{k+1} = exp_k$	(given)
3.	$e_k > 0$	(given)
4.	$e_k mod 2 \neq 0$	(given)
5.	$ans_{k+1} = ans_k \times b_k$	(given)
6.	$b_{k+1} = b_k$	(given)
7.	$e_{k+1} = e_k - 1$	(given)
8.	$ans_k \times {b_k}^{e_k} = {base_k}^{exp_k}$	(given)
9.	$ans_{k+1} \times {b_{k+1}}^{e_{k+1}} \neq {base_{k+1}}^{exp_{k+1}} \vee e_{k+1} < 0$	(negation of P_{k+1})
10.	$ans_{k+1} \times {b_{k+1}}^{e_k - 1} \neq {base_{k+1}}^{exp_{k+1}} \vee e_k - 1 < 0$	(7,9 substitution)
11.	$\left(ans_{k+1} \times {b_{k+1}}^{e_k - 1} \neq {base_{k+1}}^{exp_{k+1}} \vee e_k \leq 0\right) \wedge e_k > 0$	(3,10 conjunction)
12.	$\left(ans_{k+1} \times {b_{k+1}}^{e_k - 1} \neq {base_{k+1}}^{exp_{k+1}}\right)$	(11 disjunctive syllogism)
13.	$ans_k \times b_k \times {b_{k+1}}^{e_k - 1} \neq {base_{k+1}}^{exp_{k+1}}$	(5,12 substitution)
14.	$ans_k \times b_k \times {b_k}^{e_k - 1} \neq {base_{k+1}}^{exp_{k+1}}$	(6,13 substitution)
15.	$ans_k \times {b_k}^{e_k - 1 + 1} \neq {base_{k+1}}^{exp_{k+1}}$	(14 simplification)
16.	$ans_k \times {b_k}^{e_k} \neq {base_{k+1}}^{exp_{k+1}}$	(7,15 substitution)
17.	$ans_k \times {b_k}^{e_k} \neq {base_k}^{exp_{k+1}}$	(1,16 substitution)
18.	$ans_k \times {b_k}^{e_k} \neq {base_k}^{exp_k}$	(2,17 substitution)
19.	$ans_k \times {b_k}^{e_k} = {base_k}^{exp_k} \wedge$ $ans_k \times {b_k}^{e_k} \neq {base_k}^{exp_k}$	(8,18 conjunction)
20.	$false$	(19 contradiction)

[1]If only some paths maintain an invariant, then the loop is said to inconsistently maintain that invariant. Inconsistency is considered later in this chapter

Therefore, P_{k+1} must be implied by $P_k \wedge R_{k+1}$, and path 1 maintains the invariant $ans \times b^e = base^{exp} \wedge e \geq 0$.

Case2:

Similarly, we prove that

$$P_k \wedge R_{2,k+1} \Rightarrow (ans_{k+1} \times {b_{k+1}}^{e_{k+1}} = {base_{k+1}}^{exp_{k+1}} \wedge e_{k+1} \geq 0)$$

That is, given

$$\left(\begin{array}{l} ans_k \times {b_k}^{e_k} = {base_k}^{exp_k} \wedge e_k \geq 0 \wedge \\ base_{k+1} = base_k \wedge exp_{k+1} = exp_k \wedge e_k > 0 \wedge \\ e_k mod 2 = 0 \wedge ans_{k+1} = ans_k \wedge b_{k+1} = {b_k}^2 \wedge e_{k+1} = \frac{e_k}{2} \end{array} \right)$$

we prove that we can infer

$$ans_{k+1} \times {b_{k+1}}^{e_{k+1}} = {base_{k+1}}^{exp_{k+1}} \wedge e_{k+1} \geq 0$$

The proof is by contradiction; that is, given $P_k \wedge R_{1,k+1}$, show that supposing $ans_{k+1} \times {b_{k+1}}^{e_{k+1}} \neq {base_{k+1}}^{exp_{k+1}} \vee e_{k+1} < 0$ results in a contradiction.

1.	$base_{k+1} = base_k$	(given)
2.	$exp_{k+1} = exp_k$	(given)
3.	$e_k > 0$	(given)
4.	$e_k mod 2 = 0$	(given)
5.	$ans_{k+1} = ans_k$	(given)
6.	$b_{k+1} = {b_k}^2$	(given)
7.	$e_{k+1} = \frac{e_k}{2}$	(given)
8.	$ans_k \times {b_k}^{e_k} = {base_k}^{exp_k}$	(given)
9.	$ans_{k+1} \times {b_{k+1}}^{e_{k+1}} \neq {base_{k+1}}^{exp_{k+1}} \vee e_{k+1} < 0$	(negation of P_{k+1})
10.	$ans_{k+1} \times {b_{k+1}}^{e_{k+1}} \neq {base_k}^{exp_{k+1}} \vee e_{k+1} < 0$	(1,9 substitution)
11.	$ans_{k+1} \times {b_{k+1}}^{e_{k+1}} \neq {base_k}^{exp_k} \vee e_{k+1} < 0$	(2,10 substitution)
12.	$ans_k \times {b_{k+1}}^{e_{k+1}} \neq {base_k}^{exp_k} \vee e_{k+1} < 0$	(5,11 substitution)
13.	$ans_k \times \left(b_k^2\right)^{e_{k+1}} \neq {base_k}^{exp_k} \vee e_{k+1} < 0$	(6,12 substitution)
14.	$ans_k \times \left(b_k^2\right)^{\frac{e_k}{2}} \neq {base_k}^{exp_k} \vee \frac{e_k}{2} < 0$	(7,13 substitution)
15.	$ans_k \times {b_k}^{e_k} \neq {base_k}^{exp_k} \vee e_k < 0$	(14 simplification)
16.	$(ans_k \times {b_k}^{e_k} \neq {base_k}^{exp_k} \vee e_k < 0) \wedge e_k > 0$	(3,15 conjunction)
17.	$ans_k \times {b_k}^{e_k} \neq {base_k}^{exp_k} \wedge e_k > 0 \vee false$	(16 associativity)
18.	$ans_k \times {b_k}^{e_k} \neq {base_k}^{exp_k} \wedge e_k > 0$	(17 simplification)
19.	$ans_k \times {b_k}^{e_k} = {base_k}^{exp_k} \wedge$ $ans_k \times {b_k}^{e_k} \neq {base_k}^{exp_k} \wedge e_k > 0$	(8,18 conjunction)
20.	$false$	(19 contradiction)

Therefore, path 2 maintains the invariant $ans \times b^e = base^{exp} \wedge e \geq 0$, and so the loop maintains the invariant $P \equiv (ans \times b^e = base^{exp} \wedge e \geq 0)$.

We can reason that the loop establishes the postcondition $ans = base^{exp}$ if it terminates normally, as P and the negation of the guard ($e \leq 0$), implies $ans = base^{exp}$. That is,

$$(P \wedge e \leq 0) \Rightarrow (ans \times b^e = base^{exp} \wedge e = 0) \Rightarrow (ans = base^{exp})$$

Therefore, the loop in Program 14 is partially correct with respect to the precondition $ans = 1 \wedge b = base \wedge e = exp$ and postcondition $ans = base^{exp}$.

■

Proof by contradiction was used as it requires $\neg P_{k+1}$ be added to the list of given facts. P_{k+1} contains references to $k+1$st iteration instance variables only. The given facts, available from the conjunction of P_k and R_{k+1} describe equalities for all free variables in P_{k+1} in terms of kth iteration instances of variables. This means that we can attempt to substitute for the variables in $\neg P_{k+1}$ and find a contradiction (probably with P_k). If a top-down, direct proof strategy was employed, substitutions required to proceed toward deducing P_{k+1} would involve the identification of inverse functions to substitute $k+1$st iteration instances for kth iteration instances. A top-down strategy may, therefore, be difficult to automate. We summarises the above strategy for using a derived iterative-form invariant and a given invariant P for proving partial correctness of a loop with the following steps:

1. Generate proof obligations[2] for proving that P is maintained by the loop:
   - Base case obligation: $Q \Rightarrow P$.
   - Inductive obligation: $P_k \wedge R_{k+1} \Rightarrow P_{k+1}$.
2. Either manually or with the assistance of a theorem prover, prove the generated proof obligations. The inductive step is best dealt with by the following strategy: For every path i through the loop body, prove $P_k \wedge R_{i,k+1} \Rightarrow P_{k+1}$ by contradiction.
   (a) List every conjunct of $P_k \wedge R_{i,k+1}$ as a fact.
   (b) Suppose $\neg P_{k+1}$ and list as a fact.
   (c) Using the equalities in $R_{i,k+1}$ make substitutions in $\neg P_{k+1}$ and simplify.
   (d) Conjoin P_k and $\neg P_{k+1}$ with all substitutions made, and show that this is a contradiction
3. Show that P is a suitable invariant by proving that $P \wedge \neg G \Rightarrow R$

A formal proof was presented above to demonstrate that the derived information could support formal verification. Due to the simplicity of the steps in conducting such a proof, it is proposed that this method is readily supported by automatic and interactive theorem-provers with only simple predicate calculus rules of inference being required. Rules of inference regarding the program language are not necessary as they are defined by the predicate transformers yielding R_{k+1}.

A formal proof is not often necessary. The information described by $P_k \wedge R_{k+1}$ is just as useful for supporting a semi-formal or informal proof that $P_k \wedge R_{k+1} \Rightarrow$

[2]The base case and inductive step proof obligations can be generated automatically.

P_{k+1}. We demonstrate a semi-formal proof with the following example which illustrates that the method is constructive in that it is useful for finding defects in incorrect programs.

Program 15 `shortmerge` example

$$\left\{\begin{array}{l} p = 1 \wedge j = 1 \wedge m > 0 \wedge n > 0 \wedge \\ \forall l \in [1..m-1]\,(a[l] < a[l+1]) \wedge \forall l \in [1..n-1]\,(b[l] < b[l+1]) \end{array}\right\}$$

```
procedure shortmerge
    (a,b: IN seqℤ; c: OUT seqℤ; m: IN ℤ; j,p: INOUT ℤ)
{
    i: ℤ; i := 1;
```

$Q =$

$$\left(\begin{array}{l} j = j? = 1 \wedge m = m? \wedge n = n? \wedge m > 0 \wedge n > 0 \wedge p = p? = 1 \wedge i = 1 \wedge \\ a = a? \wedge b = b? \wedge a, b, c \in \texttt{seq}\ \mathbb{Z} \wedge \\ \forall l \in [1..m-1]\,(a[l] < a[l+1]) \wedge \forall l \in [1..n-1]\,(b[l] < b[l+1]) \end{array}\right)$$

```
    do i ≤ m →
        if a[i] ≤ b[j] →
            c[p] := a[i];
            p := p + 1;
            i := i + 1;
        [] b[j]<a[i] →
            c[p] := b[j];
            p := p + 1;
            j := j + 1;
        fi;
    od;
}
```

The procedure `shortmerge`, in Program 15, is part of a *mergesort* algorithm presented in [25].

The invariant for `shortmerge` is given as:

> After an arbitrary iteration, $a[1..i-1]$ is merged with $b[1..j-1]$ and stored in $c[1..p-1]$ and c is sorted in non-descending order and $i \leq m+1 \wedge j \leq n+1 \wedge p = i+j-1$.

We denote this invariant P and define formally by

$$P = \left(\begin{array}{l} \forall l \in [1..p-2]\,(c[l] \leq c[l+1]) \wedge \quad --- \quad \text{sorted} \\ \forall l \in [1..p-1]\left(\#\{x \in [1..p-1] | c[l] = c[x]\} = \left(\begin{array}{l} \#\{x \in [1..i-1] | c[l] = a[x]\} + \\ \#\{x \in [1..j-1] | c[l] = b[x]\} \end{array}\right)\right) \wedge \\ i \leq m+1 \wedge j \leq n+1 \wedge p = i+j-1 \end{array}\right)$$

By theorem 9.2, partial correctness is proven as follows.

Proof. **Base Case:**

Prove that $Q \Rightarrow P$, where

$$Q = \begin{pmatrix} j = j? = 1 \wedge m = m? \wedge n = n? \wedge m > 0 \wedge n > 0 \wedge p = p? = 1 \wedge i = 1 \wedge \\ a = a? \wedge b = b? \wedge a, b, c \in \textbf{seq } \mathbb{Z} \wedge \\ \forall l \in [1..m-1]\,(a[l] < a[l+1]) \wedge \forall l \in [1..n-1]\,(b[l] < b[l+1]) \end{pmatrix}$$

Given that Q is true, we know $p = p? = 1 \wedge j = j? = 1 \wedge i = 1$, and we can simplify P as follows:

$$\begin{aligned} P \quad &\equiv \quad \begin{pmatrix} \forall l \in [1..1-2]\,(c[l] \leq c[l+1]) \wedge \\ \forall l \in [1..1-1] \left(\# \{x \in [1..1-1] | c[l] = c[x]\} = \begin{pmatrix} \# \{x \in [1..1-1] | c[l] = a[x]\} + \\ \# \{x \in [1..1-1] | c[l] = b[x]\} \end{pmatrix} \right) \wedge \\ 1 \leq m+1 \wedge 1 \leq n+1 \wedge 1 = 1 \end{pmatrix} \\ &\equiv \quad \begin{pmatrix} true \wedge true \wedge \\ 1 \leq m+1 \wedge 1 \leq n+1 \wedge true \end{pmatrix} \\ &\equiv \quad true \text{ (Because } m > 0 \wedge n > 0) \end{aligned}$$

Therefore, $Q \Rightarrow P$, and P is maintained on initialisation.

Inductive Step:

Assume P_k, and prove $sp\,(P_k \wedge Q_{k+1}, S) \Rightarrow P_{k+1}$, where P_k is given by

$$P_k = \begin{pmatrix} \forall l \in [1..p_k - 2]\,(c[l]_k \leq c[l+1]_k) \wedge \\ \forall l \in [1..p_k - 1] \begin{pmatrix} \# \{x \in [1..p_k - 1]\,| c[l]_k = c[x]_k\} = \\ \begin{pmatrix} \# \{x \in [1..i_k - 1]\,| c[l]_k = a[x]_k\} + \\ \# \{x \in [1..j_k - 1]\,| c[l]_k = b[x]_k\} \end{pmatrix} \end{pmatrix} \wedge \\ i_k \leq m_k + 1 \wedge j_k \leq n_k + 1 \wedge p_k = i_k + j_k + 1 \end{pmatrix}$$

and P_{k+1} is given by

$$P_{k+1} = \begin{pmatrix} \forall l \in [1..p_{k+1} - 2]\,(c[l]_{k+1} \leq c[l+1]_{k+1}) \wedge \\ \forall l \in [1..p_{k+1} - 1] \begin{pmatrix} \# \{x \in [1..p_{k+1} - 1]\,| c[l]_{k+1} = c[x]_{k+1}\} = \\ \begin{pmatrix} \# \{x \in [1..i_{k+1} - 1]\,| c[l]_{k+1} = a[x]_{k+1}\} + \\ \# \{x \in [1..j_{k+1} - 1]\,| c[l]_{k+1} = b[x]_{k+1}\} \end{pmatrix} \end{pmatrix} \wedge \\ i_{k+1} \leq m_{k+1} + 1 \wedge j_{k+1} \leq n_{k+1} + 1 \wedge p_{k+1} = i_{k+1} + j_{k+1} + 1 \end{pmatrix}$$

and we calculate

$$R_{k+1} \equiv \begin{pmatrix} k \in \mathbb{N} \wedge i_{k+1} = i_k \wedge \\ a_{k+1} = a_k \wedge b_{k+1} = b_k \wedge m_{k+1} = m_k \wedge n_{k+1} = n_k \wedge p_{k+1} = p_k + 1 \wedge \\ \begin{pmatrix} \left(a\,[i_k]_{k+1} \leq b\,[j_k]_{k+1} \wedge c\,[p_k]_{k+1} = a\,[i_k]_{k+1} \wedge j_{k+1} = j_k \wedge i_{k+1} = i_k + 1 \wedge i_k \leq m\right) \vee \\ \left(b\,[j_k]_{k+1} < a\,[i_k]_{k+1} \wedge c\,[p_k]_{k+1} = b\,[j_k]_{k+1} \wedge j_{k+1} = j_k + 1 \wedge i_{k+1} = i_k \wedge i_k \leq m\right) \end{pmatrix} \end{pmatrix}$$

We will prove the inductive hypothesis in three steps. We will start by proving that the sorted property of c is maintained after the $k+1$st iteration. That is, we prove

$$(P_k \wedge R_{k+1}) \Rightarrow \forall l \in [1..p_{k+1} - 2]\,(c[l]_{k+1} \leq c[l+1]_{k+1})$$

$\forall l \in [1..p_k - 2]\,(c[l]_k \leq c[l+1]_k)$ holds due to P_k. From R_{k+1}, we see that only the element at the index p_k is changed, so $\forall l \in [1..p_k - 1]\,(c[l]_k = c[l]_{k+1})$. Therefore, $\forall l \in [1..p_k - 2]\,(c[l]_{k+1} \leq c[l+1]_{k+1})$ holds. Now, $c\,[p_k]_{k+1}$ is equal to the lesser of $a\,[i_k]_{k+1}$ and $b\,[j_k]_{k+1}$. The sequences a and b are originally sorted in ascending order, and from R_{k+1} we can see that a and b are not changed by the loop. We can also reason that after the kth iteration c is composed of $a[1..i_k - 1]$ and $b[1..j_k - 1]$, so concatenating the lesser of $a\,[i_k]$ and $b\,[j_k]$ to c on this iteration concatenates the next largest element from a and b to c, and hence, $c\,[p_k - 1] \leq c\,[p_k]$. This will maintain the sorting property, and $\forall l \in [1..p_{k+1} - 2]\,(c[l]_{k+1} \leq c[l+1]_{k+1})$ will hold.

Next we prove that the c is composed only of elements from a and b and that those elements are not appended to c more than once. That is we prove

$$(P_k \wedge R_{k+1}) \Rightarrow$$
$$\forall l \in [1..p_{k+1} - 1] \left(\begin{array}{l} \#\{x \in [1..p_{k+1} - 1]\,|c[l]_{k+1} = c[x]_{k+1}\} = \\ \left(\begin{array}{l} \#\{x \in [1..i_{k+1} - 1]\,|c[l]_{k+1} = a[x]_{k+1}\} + \\ \#\{x \in [1..j_{k+1} - 1]\,|c[l]_{k+1} = b[x]_{k+1}\} \end{array} \right) \end{array} \right)$$

From P_k we are given

$$\forall l \in [1..p_k - 1] \left(\begin{array}{l} \#\{x \in [1..p_k - 1]\,|c[l]_k = c[x]_k\} = \\ \left(\begin{array}{l} \#\{x \in [1..i_k - 1]\,|c[l]_k = a[x]_k\} + \\ \#\{x \in [1..j_k - 1]\,|c[l]_k = b[x]_k\} \end{array} \right) \end{array} \right)$$

Now, either $c\,[p_k]_{k+1} = a\,[i_k]_{k+1}$ and the indexes for a and c are increased by one, or $c\,[p_k]_{k+1} = b\,[j_k]_{k+1}$ and the indexes for b and c are increased by 1. The next element of either a or b is appended to c, and because the indexes are incremented (and never decremented) then elements of a and b are only appended once. The index p for c is monotonically increased, and so past elements are not overwritten. Therefore,

$$\forall l \in [1..p_k - 1] \left(\begin{array}{l} \#\{x \in [1..p_{k+1} - 1]\,|c[l]_{k+1} = c[x]_{k+1}\} = \\ \left(\begin{array}{l} \#\{x \in [1..i_{k+1} - 1]\,|c[l]_{k+1} = a[x]_{k+1}\} + \\ \#\{x \in [1..j_{k+1} - 1]\,|c[l]_{k+1} = b[x]_{k+1}\} \end{array} \right) \end{array} \right)$$

and $c\,[p_k]_{k+1} = a\,[i_k]_{k+1}$ and $p_{k+1} = p_k + 1 \wedge i_{k+1} = i_k + 1$ or

$$\forall l \in [1..p_k - 1] \left(\begin{array}{l} \#\{x \in [1..p_{k+1} - 1]\,|c[l]_{k+1} = c[x]_{k+1}\} = \\ \left(\begin{array}{l} \#\{x \in [1..i_{k+1} - 1]\,|c[l]_{k+1} = a[x]_{k+1}\} + \\ \#\{x \in [1..j_{k+1} - 1]\,|c[l]_{k+1} = b[x]_{k+1}\} \end{array} \right) \end{array} \right)$$

and $c\,[p_k]_{k+1} = b\,[j_k]_{k+1}$ and $p_{k+1} = p_k + 1 \wedge j_{k+1} = j_k + 1$

It follows that

$$\forall l \in [1..p_{k+1} - 1] \left(\begin{array}{l} \#\{x \in [1..p_{k+1} - 1]\,|c[l]_{k+1} = c[x]_{k+1}\} = \\ \left(\begin{array}{l} \#\{x \in [1..i_{k+1} - 1]\,|c[l]_{k+1} = a[x]_{k+1}\} + \\ \#\{x \in [1..j_{k+1} - 1]\,|c[l]_{k+1} = b[x]_{k+1}\} \end{array} \right) \end{array} \right)$$

is maintained.

Finally we need to prove

$$P_k \wedge R_{k+1} \Rightarrow i_{k+1} \leq m+1 \wedge j_{k+1} \leq n+1 \wedge p_{k+1} \leq i_{k+1}+j_{k+1}+1$$

We know $i_k \leq m+1 \wedge j_k \leq n+1 \wedge p_k \leq m+n+1$ from P_k. Additionally R_{k+1} implies that $i_k \leq m$ and $i_{k+1} = i_k + 1$, and so, $i_{k+1} \leq m+1$ holds.

The inequality, $j_{k+1} \leq n+1$, may not be maintained after the $k+1$st iteration.

From $R_{1,k+1}$, $j_{k+1} = j_k$ is given, and $j_k \leq n+1$ is given by P_k. By substitution, we conclude $P_k \wedge R_{1,k+1}$ establishes $j_{k+1} \leq n+1$, and so path 1 maintains $j \leq n+1$.

From $R_{2,k+1}$, $j_{k+1} = j_k + 1$ is given, and $j_k \leq n+1$ is given by P_k. By substitution, we conclude $P_k \wedge R_{2,k+1}$ establishes $j_{k+1} \leq n+2$. This does not imply $j_{k+1} \leq n+1$, and so path 2 does not maintain the given invariant. That is, if $j_k = n+1$ after the kth iteration, then after the $k+1$st iteration, $j_{k+1} = n+2$ will hold after iteration through path 2, and the invariant $j_{k+1} \leq n+1$ is false. Furthermore, on the $k+1$st iteration, $b[n+1]$ will be compared to $a[i_k]$. The fact that $b[n+1]$ is referenced indicates that the loop has iterated 1 too many times. We can reason that this will occur whenever $b[n] < a[m]$. The given precondition, does not exclude this situation from occurring, and so `shortmerge` is not correct with respect to the given precondition and invariant, and therefore the given postcondition.

From $R_{1,k+1}$, we are given $p_{k+1} = p_k + 1 \wedge i_{k+1} = i_k + 1 \wedge j_{k+1} = j_k$, and $P_k \Rightarrow p_k = i_k + j_k + 1$. By substitution, we conclude $p_{k+1} - 1 = i_{k+1} - 1 + j_{k+1} + 1 \equiv p_{k+1} = i_{k+1} + j_{k+1} + 1$. Therefore path 1 maintains the invariant $p = i+j+1$. Similarly we prove that $R_{2,k+1} \wedge P_k \Rightarrow p_{k+1} = i_{k+1} + j_{k+1} + 1$, and so path 2 maintains $p = i+j+1$, therefore the loop maintains $p = i+j+1$.

In summary

$$\left(\begin{array}{l} \forall l \in [1..p-2]\,(c[l] \leq c[l+1]) \wedge \\ \forall l \in [1..p-1]\left(\#\{x \in [1..p-1] | c[l] = c[x]\} = \left(\begin{array}{l} \#\{x \in [1..i-1] | c[l] = a[x]\} + \\ \#\{x \in [1..j-1] | c[l] = b[x]\} \end{array}\right)\right) \wedge \\ i \leq m+1 \wedge p = i+j+1 \end{array}\right)$$

is maintained, but $j \leq n+1$ is not consistently maintained. □

This example shows a failed proof of correctness conducted using only mechanically derived conditions and a given invariant. By conducting such a proof the cause of defects can be identified. In this example we have identified an intended invariant that is not consistently maintained. In the next section we discuss further the identification of inconsistencies.

9.1.2 Identifying Inconsistency

Definition 12. A loop is said to be **inconsistent** with respect to a given invariant, P, if at least one path through the body of the loop does not maintain P.

Given a loop, `do G → S od`, and invariant P, where the iterative form invariant is calculated as $R_{k+1} \equiv R_{1,k+1} \vee \cdots \vee R_{n.k+1}$, where n is the number of paths through the loop body, S. We formally define inconsistency by (9.8).

$$\exists i \in [1..n]\,(\neg\,(P_k \wedge R_{i,k+1} \Rightarrow P_{k+1})) \tag{9.8}$$

Theorem 9.3. [Invariant Inconsistency Theorem]

If $\exists i \in [1..n]\,(\neg\,(P_k \wedge R_{i,k+1} \Rightarrow P_{k+1}))$, then the loop is inconsistent with respect to P, and for every i that satisfies (9.8), path i does not maintain P.

Proof. This is true by the above definition, and by the fact that, by calculation, every disjunct, $R_{i,k+1}$, of R_{k+1} corresponds to a path, i, through the loop body. □

This technique is constructive in that it identifies which paths maintain P and which don't. Such information is useful for the correction of defects associated with this inconsistency.

9.1.3 Direct Method

If a strong, conjunctive form of invariant, and hence postcondition, is able to be derived using the techniques of chapters 5 and 6, then formal proof is possible without applying induction and without requiring that an invariant be given.

For any loop, `do G → S od`, if we can derive a strong, conjunctive postcondition, R under a given precondition Q, then in order to reason that that loop establishes a given postcondition, R_{given} under Q, we must show that

$$R \Rightarrow R_{given} \tag{9.9}$$

For example, consider the loop in Program 8 for finding the first occurrence of the maximum value in an array, for which we derived the following postcondition (see chapter 6), under the precondition *true*:

$$R \equiv \left(\begin{array}{l} i \in \mathbb{N}^+ \wedge i = N \wedge 0 \leq p \leq i \wedge \\ \forall j \in [0..p-1]\,(a\,[j] < a[p]) \wedge \forall j \in [p..i-1]\,(a\,[j+1] \leq a[p]) \end{array} \right)$$

Let the given postcondition, denoted R_{given}, for this loop be:

$$R_{given} \equiv 0 \leq p \leq N \wedge \forall j \in [0..p-1]\,(a\,[j] < a\,[p]) \wedge \forall j \in [p+1..N]\,(a\,[j] \leq a\,[p])$$

We can reason that $R \Rightarrow R_{given}$. Therefore, by (9.9), we say that the find first maximum example (Program 8) is partially correct with respect to the given pre and postcondition.

This method is reliant on our ability to derive and use a strong invariant and postcondition from the program.

9.2 Function-Theoretic Correctness Verification Method

Correctness verification in Cleanroom software development involves showing that a procedure implements its specified *intended function.* In the function-theoretic approach, a procedure is correct with respect to its specification when every control structure in that procedure is verified to do what its intended function specifies[86].

The correctness condition for iterative structures is given in the following question form in [86][3], where $[f]$ is a clear-box specification for the loop.

$[f]$

`do` $p \rightarrow$

g

`od`

Correctness Conditions:

1. Does the loop terminate?
2. When p is true, does g followed by f do f.
3. When p is false, does doing nothing do f.

Question 1 can be answered by reasoning about information generated from chapter 7.

As an example of answering question 2, consider the binary search example in Program 16 to find an employee's id. in an array of employees (translated into our modified guarded commands language from [86]).

The correctness questions require that the loop in the binary search satisfies the intended function, `[if possible, set` i `so that` $emp(i) = id$ `and` $b \leq i \leq t$`, otherwise leave` i `unchanged]`. The proof of this, as defined by the function-theoretic approach is as follows.

The given intended function for the loop body specifies the intended net effect of the loop body. That is, the loop body has the intended function,g,

`[if` $id = emp((b+t)/2)$`, set` i `to` $(b+t)/2$`, otherwise`
`if` $id > emp((b+t)/2)$`, set` b `to` $(b+t)/2$`, otherwise`
`if` $id < emp((b+t)/2)$`, set` t `to` $(b+t)/2$`, otherwise ]`

In order to answer the second question, we verify that the intended function for the loop body, g, followed by the intended function for the loop, f,

[3]The form of the structure has been translated to use Guarded Commands.

Program 16 Employee Binary Search Example Loop

$\{i = 0 \wedge b = 1 \wedge t = n \wedge emp \in \texttt{seq}\ \mathbb{N} \wedge id \in \mathbb{N}\}$

[if possible, set i so that $emp(i) = id$ and $b \leq i \leq t$, otherwise
leave i unchanged]

do b≤t ∧ i=0 →

[if $id = emp((b+t)/2)$, set i to $(b+t)/2$, otherwise
if $id > emp((b+t)/2)$, set b to $(b+t)/2$, otherwise
if $id < emp((b+t)/2)$, set t to $(b+t)/2$, otherwise]

```
    m := (b+t)/2;
    if emp[m]=id →
        i:=m;
    [] emp[m]<id →
        b := m+1;
    [] emp[m]>id →
        t := m+1;
    fi;
od
```

$$R_{k+1} \equiv \left(\begin{array}{l} b_k \leq t_k \wedge i_k = 0 \wedge \\ \left(\begin{array}{l} (b_{k+1} = b_k \wedge t_{k+1} = t_k \wedge m_{k+1} = (b_k + t_k)/2 \wedge emp[m]_{k+1} = id \wedge i_{k+1} = m_{k+1}) \vee \\ (b_{k+1} = m_{k+1} \wedge t_{k+1} = t_k \wedge m_{k+1} = (b_k + t_k)/2 \wedge emp[m]_{k+1} < id \wedge i_{k+1} = i_k) \vee \\ (b_{k+1} = b_k \wedge t_{k+1} = m_{k+1} \wedge m_{k+1} = (b_k + t_k)/2 \wedge emp[m]_{k+1} > id \wedge i_{k+1} = i_k) \end{array} \right) \end{array} \right)$$

[if $id = emp((b+t)/2)$, set i to $(b+t)/2$, otherwise
if $id > emp((b+t)/2)$, set b to $(b+t)/2$, otherwise
if $id < emp((b+t)/2)$, set t to $(b+t)/2$, otherwise]
[if possible, set i so that $emp(i) = id$ and $b \leq i \leq t$, otherwise
leave i unchanged]

satisfies the intended function for the loop, f,

[if possible, set i so that $emp(i) = id$ and $b \leq i \leq t$, otherwise
leave i unchanged]

The details of this verification are given in [86].

In order for this verification to be trusted, we must then show that the loop body establishes the intended function g. Using the iterative-form invariant, we reason that R_{k+1} describes the intended function for the loop body, g.

A simpler approach to answering the second correctness question, involves using the derived iterative-form invariant, R_{k+1}, to represent the net effect of executing the loop body. That is, we can use R_{k+1} itself in place of the loop body in the verification, and show that R_{k+1} followed by the intended function f satisfies f. The benefit of using R_{k+1} is that R_{k+1} is a direct representation of the loop body, obtained by direct calculation, therefore, no verification is required to show that the loop body establishes R_{k+1}. The following proof reasoning is given to show that R_{k+1} followed by f does f for the binary search example.

Proof. Consider the first disjunct

$$b_k \leq t_k \wedge i_k = 0 \wedge b_{k+1} = b_k \wedge t_{k+1} = t_k \wedge m_{k+1} = (b_k + t_k)/2 \wedge emp[m]_{k+1} = id \wedge i_{k+1} = m_{k+1}$$

We see that after an arbitrary $k+1$st iteration, $emp[m] = id$ and that $i = m = (b+t)/2$, so $emp[i] = id$ and $b \leq i \leq t$. By the correctness condition 2, we show that the composition

$$\begin{array}{l} emp[i] = id \wedge b \leq i \leq t \\ [f] \end{array}$$

does f. We do this by reasoning that this path leads to id being found at $emp(i)$, and so it is possible to set i so that $emp(i) = id$ where $b \leq i \leq t$.

We must also show that if id is not found on an arbitrary $k+1$st iteration, then i is not changed. The composition of the second and third disjuncts with the intended function, gives

$$\begin{array}{l} (emp[(b+t)/2] < id \wedge i = 0) \vee (emp[(b+t)/2] > id \wedge i = 0) \\ [f] \end{array}$$

We reason that this does the intended function, because i is unchanged and future iterations will search the part of emp where id may be found. □

The third verification question, is directly answerable by showing that the negation of the guard satisfies the intended function of the loop. In this case, that $b > t \vee i \neq 0$ satisfies `[if possible, set` i `so that` $emp(i) = id$ `and` $b \leq i \leq t$`, otherwise leave` i `unchanged]`. We note from the above that i only changes when id is found in emp. In this case, i is set so that $emp(i) = id \wedge b \leq i \leq t$. If i is not found, then $i = 0$ is maintained, and $b > t$ is satisfied when emp has been searched. Therefore,`[if possible, set` i `so that` $emp(i) = id$ `and` $b \leq i \leq t$`, otherwise leave` i `unchanged]` is satisfied by doing nothing when the guard is false.

9.3 Conclusion

In this chapter, three methods of formal verification have been presented. The contribution of this chapter is that it provides a strategy for using the information generated by the algorithms in the previous chapters for supporting both semi-formal and formal verification. It has been shown that verification can be conducted using these methods without reference to the program itself. The prototype tool can generate the required proof obligations for the inductive axiomatic method and the function-theoretic method if presented with a suitable invariant. If a strong, conjunctive postcondition can be generated by the tool it is also capable of presenting the direct proof obligations. Appendix B shows some examples of the generated proof obligations. These proof obligations are presented as first order logic formulae which may be proved either manually by hand or automatically/interactively with the assistance of a theorem prover.

Chapter 10

A Prototype Tool

A prototype tool has been developed to implement a number of the algorithms discussed in this dissertation. This prototype serves two purposes: (i) to demonstrate that the algorithms defined are mechanisable and the approach is viable, and (ii) to remove the burden of the tedious parts of the derivation process from the verifier, and the author in working the examples of this thesis. Some output from the prototype for many of the examples in this thesis is presented in Appendix B.

The prototype currently automates the process of deriving strongest postconditions from procedures containing no iterative constructs (see sections B.9 and B.10). From programs with iterative constructs, the prototype will always yield an iterative-form invariant. For non-branching programs yielding difference equations in one of the forms in table 5.1, the prototype will generate a non-iterative invariant and postcondition (see sections B.1,B.4 and B.7). The prototype also yields universally quantified formulae describing iterative programs which deal with non-branching loops with simple array assignments (see sections B.5, B.6 and B.8).

10.0.1 The Architecture

The architecture of the prototype tool is illustrated in Fig. 10.1. The prototype generates standard form abstract syntax trees (ASTs) for programs written in a supported language (currently only a variant of Dijkstra's Guarded Commands Language is supported). This standard form is based on the principle that all imperative languages have composition constructs for sequence, selection and iteration, and statements for handling assignment, procedure call and IO. For example, an iterative construct in a particular language will yield the same AST node as a similar iterative construct in another language. This allows us to add semantic analysis rules (generally implementations of the *sp* predicate transformer) to the system to deal only with constructs, independent of language. This architecture is extensible for other languages by adding a parser for the language which produces standard form ASTs with new node types for

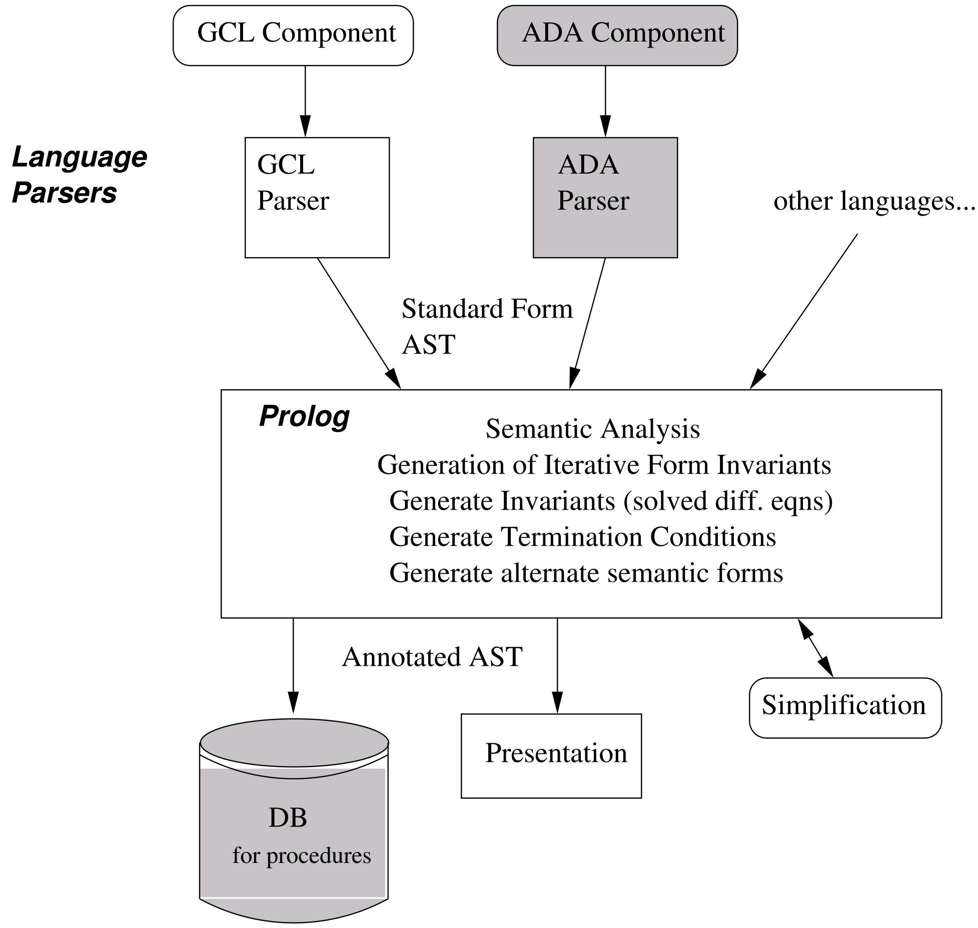


Figure 10.1: The architecture of the prototype tool under development (greyed sections are planned for the future).

language specific constructs and elements, and by adding rules to the semantic analyser to deal with any new AST node types.

For example, [39] defines strongest postcondition calculation rules to handle C pointers and simple pointer arithmetic. If the prototype is to be extended to handle C, a parser for the chosen C grammar would need to be constructed, which produces special AST node types to represent pointer operations. The semantic analyser part of the system would then need extension to calculate the strongest postcondition from statements involving pointers. That is, it would need to implement the rules defined in [39] to deal with pointer node types in the AST.

We see this approach as a viable solution to the problem of dealing with many languages. Language parsers are easy to build with tools such as LEX and YACC, in fact many grammars for languages are freely available. The grammars are simply extended to generate ASTs composed of the standard AST nodes.

The semantic analyser comprises a number of components implemented in *Prolog*. Prolog was chosen for this task as many of the algorithms presented are dependent on pattern matching and substitution, and Prolog lends itself to the implementation of such algorithms. Abstract syntax trees generated by the parser component are easily translated into Prolog facts. These facts are operated on by a number of components.

Firstly, the strongest postcondition is calculated for all statements and compositions of statements. For all constructs, other than iterative constructs, this calculation is direct. For iterative constructs, an iterative-form invariant is calculated as described in chapter 4. In order to save on screen/page real-estate, when displaying the iterative-form invariant the tool presents the $k+1$st iteration instances in the iterative-form invariant without the $k+1$ subscripts. That is, $v_{k+1} = v_k - 2$ would be output as $v = v_k - 2$.

For loops with non-branching loop bodies, a difference equation solver is then applied to the iterative-form invariant to solve all iterative-form difference equations in the invariant. All difference equations matching the forms in table 5.1 are solved at present. The collection of solvable equations can be extended by simply adding rules for solving other forms of difference equations. Difference equations are solved in order of increasing variable dependence. The difference equation solutions are then substituted into the iterative-form invariant in an implementation of refinement (5.23) in chapter 5. If all difference equations are solvable, then (5.26) is applied and the postcondition is presented. Failure to calculate a difference equation solution will result in the prototype abandoning the attempt to calculate a non-iterative-form invariant.

Many of the heuristics for deriving invariants for MBS loops rely on complex pattern matching and simplification that the prototype doesn't presently handle. Some of these rules are proving difficult, although we believe, not impossible, to implement. For MBS loops, only the iterative-form invariant is calculated to support manual application of the heuristic rules in Chapter 6, if an invariant and postcondition are required. The tool was used in this way to support the Dutch National Flag example in Chapter 6.

If an invariant in first order logic has been specified for a loop, the prototype always generates the required conditions for proof of partial correctness by the inductive method of Chapter 9 (see sections B.1 and B.2).

If an intended function has been specified for a loop, the prototype always generates the required conditions for proof of partial correctness by the function theoretic method of Chapter 9. The case study below is an example of this.

If a postcondition is provided and a postcondition can be extracted from the program, then the conditions for direct proof of partial correctness are also produced by the prototype (see sections B.1, B.4, B.5, B.6 and B.7). The lack of logical and algebraic simplification rules in the tool are evident in these examples. We contend, however, that the results are still useful as the manual simplification required presents no significant challenge.

A component is also implemented to calculate loop progress conditions by the non-variant function method of chapter 7. This is achieved by determining the two sides of the negated guard equality, and then simplifying the disjunction of the ten progress conditions for each disjunct of the iterative invariant. The tool reports these progress condition (see all examples in Appendix B). If an invariant can be generated the prototype also reports on the conditions required for final termination as described by Theorem 7.4.

An earlier version of the prototype also generated and presented input-output mappings directly from the iterative invariant as explained in chapter 4. Output from this version of the tool can be seen in Fig. 4.3.

The language parsed by the prototype is described in Appendix A. It provides for the specification of preconditions, postconditions and loop invariants (or intended functions). For example, the procedure corresponding to the exponentiation example in chapter 9 is given (with comments) as

```
procedure exp2(base:Int, exp:Int, ans:Int)
[base>0 && exp>0]  // precondition
{
  ans := 1;
  b := base;
  e := exp;
  do e > 0 ->
    {ans*b^e = base^exp}  // loop invariant
    if e mod 2 != 0 ->
        ans := ans*b;
        e := e-1;
    [] e mod 2 = 0 ->
        b := b*b;
        e := e div 2;
    fi;
  od;
}
[ans=base^exp];  // postcondition
```

A number of versions of the tool exist, each producing a different type of output. The current version produces LaTeX reports. Earlier versions were also written to produce text, HTML and MATHML reports. The current version is also capable of storing procedure specifications in a Prolog database to assist in implementing the strongest postcondition for procedure calls as defined in Chapter 8 (this has not yet been implemented).

The following table summarises the extent of implementation achieved in the current prototype for the algorithms, heuristics, functions and rules defined in this dissertation.

Extent of Prototype Implementation

Feature	**Implemented**	**Comment**
Free variable function	complete	
Substitution functions	complete	
Algorithm 1 - *sp* for assignment	complete	

Extent of Prototype Implementation

Feature	**Implemented**	**Comment**
sp for sequence	complete	To Dijkstra's definition
sp for selection	complete	To Dijkstra's definition
Equation (3.2) - *sp* for read	complete	
Equation (3.3) - *sp* for write	complete	
Equation (3.4) - *sp* for variable declaration	complete	
Algorithm 2 for calculating Q_{k+1}	complete	
Equation (4.2) for calculating R_{k+1}	complete	
Algorithm 3 for calculating variable maps	Incomplete	Implemented for I/O mapping in original version. Planned implementation in current version to pass names of variables to map as arguments to semantic analyser.
Difference equation solver	complete *	Difference equation solutions for all forms in Table 5.1 implemented.
Refinement (5.23) for refining R_{k+1} with difference equation solutions	complete	
Substitution (5.24) to remove references to k	not implemented	Implementation planned - requires simplification and equation solving.
Equation (5.26) for using invariant to calculate postcondition	complete	Rule only applied when invariant can be generated.
Generation of partial correctness proof obligation based on inspection rule 2	complete	Rule only applied when postcondition can be generated.
Equation (5.28) for handling non-redundant array assignments	complete	
Algorithms and heuristics for handling redundant array assignments (in Section 5.4)	incomplete	Requires simplification and equation solving not yet implemented. Some basic heuristics currently implemented but not used.
Algorithm 7 for generating invariants from non-branching loops	incomplete	Steps 4-9 not implemented (see previous row for reason). All other steps implemented. Continues with iterative-invariant if invariant can't be generated because of missing steps.
Algorithm 8 - strategy for constructing invariants from MBS loops	incomplete	Only step 1 is implemented. The remaining steps require heuristics be implemented, only some of which are implemented to date. The current prototype doesn't attempt to apply any of these implemented heuristic rules on their own.

Extent of Prototype Implementation

Feature	**Implemented**	**Comment**
Implementation of Theorem 7.3 to calculate loop progress conditions without a variant	complete	Simplification is limited.
Implementation of Theorem 7.4 to define termination conditions	incomplete	Complete for loops for which the tool can generate a conjunctive invariant (ie. non-branching loops).
Algorithm 12 for abstracting procedure semantics	incomplete	All parameters are currently treated as reference parameters.
Algorithm 13 for checking preconditions of procedure calls and Algorithm 14 for calculating *sp* for procedure call	not implemented	Should be simple as they are based on substitution.
Algorithms for abstracting function semantics and calculating *sp* for function calls	not implemented	GCL Parser does not yet recognise functions.
Generation of proof obligations from axiomatic verification method (given a first order logic invariant)	complete	Tool also stores proof obligation as a Prolog fact. This may be useful as a basis for interacting with a Prolog based theorem prover such as Ergo[91].
Generation of proof obligations from function-theoretic verification method (given an invariant or intended function in natural language)	complete	
Generation of partial correctness proof obligations using the direct method (given a postcondition)	complete *	Only generated when a postcondition is given and one is calculable from the code.

10.1 A Case Study Using the Tool

As an example of using the tool to support verification based inspection with a program, consider a program to find the length of the longest monotone increasing sub-sequence of an array of integers. A monotone increasing sub-sequence is a subset of numbers which are strictly increasing from left to right. In [25] a program and specification are presented for solving this problem. In the program Fig 10.2 we have attempted a translation of this program to an implementation which does not use nested loops. This is necessary at present, as this thesis does not describe mechanisms for handling nested loops. We use the prototype to produce a report which will allow us to investigate correctness of this algorithm with respect to the given specification[1]. We note that the given specification is not written in a formal language, so an automated or automatically assisted theorem proving technique applied to this problem would first require a formalising of the given specification. We use this example to illustrate that one of the main benefits of supporting verification based inspection is that human verification is possible even with a semiformal specification.

```
procedure monotone (a : seq Int, n : Int,maxlen : Int )
[n > 0]
{
    len: seq Int;
    i: Int; j: Int; maxj: Int; pmax: Int;
    len [1] := 1; pmax := 1; maxlen := 1; i := 2; j := 1; maxj:=1;

    do i != n ->

       {'len[1..i-1] are lengths of lmss ending at a[1..i-1], respectively, 1=<i=<n+1,
       maxlen = length of lmss in first i-1 positions, and pmax is where it ends.
       (When 1=<i=<n, maxj is length of lmss in a[1..j] terminating in a value less
       than a[i] and 1=<j=<i)'}

        if a[i] < a[pmax] && j < i && a[j + 1] < a [i] && maxj < len[j + 1] ->
            maxj := len[j + 1];
            j := j + 1;
        [] a[i] < a[pmax] && j < i && a[j + 1] < a[i] && maxj >= len[j + 1] ->
            j := j + 1;
        [] a [i] < a [pmax] && j < i && a[i] <= a[j+1] ->
            j := j + 1;
        [] a[i] < a[pmax] && j >= i && (maxj + 1) > maxlen ->
            len[i] := maxj + 1;
            maxj := 1; maxlen := (maxlen + 1); pmax := i;
            i := i + 1; j := 1;
        [] a[i] < a[pmax] && j >= i && (maxj + 1) <= maxlen ->
           len[i] := maxj + 1;
            maxj := 1;
           i := i + 1; j := 1;
        [] a[i] >= a[pmax] ->
           maxlen := maxlen + 1;
           len[i] := maxlen;
           pmax := i;
           i := i + 1; maxj := 1;
           j := 1;
        fi;
    od;
};
```

Figure 10.2: Actual parsed GCL like code for translation of find longest monotone sub-sequence

The program in Fig. 10.2 is parsed by the GCL parser resulting in the generation of an Prolog representation of the abstract syntax tree (see Fig. 10.3). This representation presents the structure of every procedure in the input file as

[1]Emphasising the benefits of formal verification, during the proof stage of this case study, the specification of the invariant given in [25] was found to be incorrect. The invariant was corrected for the case study.

a fact. The generated Prolog file is consulted and loads all of the semantic analysis libraries with the `load` command on the first line. The goal `start` is queried. This opens a file for the formatted output (in the case of this version a LaTeX file is generated). To satisfy the subgoal `process` an attempt is made to semantically analyse all of the abstract syntax trees for procedures in the file, creating a LaTeX document with a section for each.

```
:-[load].

procedure(monotone, [n>0], [dec(a,type(seq('Int'))), dec(n,type('Int')), dec(
maxlen,type('Int'))], [dec(len, type(seq('Int'))), dec(i,type('Int')), dec(j,
type('Int')), dec(maxj,type('Int')), dec(pmax,type('Int')),array(len,1):=1,pm
ax:=1,maxlen:=1,i:=2,j:=1,maxj:=1,do(intendedFun('len[1..i-1] are lengths of lmss end
ing at a[1..i-1], respectively, 1=<i=<n+1, maxlen = length of lmss in first i
-1 positions, and pmax is where it ends. (When 1=<i=<n, maxj is length of lms
s in a[1..j] terminating in a value less than a[i] and 1=<j=<i)'),i=<n,[if([a
rray(a,i)<array(a,pmax),j<i,array(a,j+1)<array(a,i),maxj<array(len,j+1)],[max
j := array(len,j+1),j:=j+1],[array(a,i)<array(a,pmax),j<i,array(a,j+1)<array(
a,i),maxj>=array(len,j+1)],[j:=j+1],[array(a,i)<array(a,pmax),j<i,array(a,i)=
<array(a,j+1)],[j:=j+1],[array(a,i)<array(a,pmax),j>=i,maxj+1>maxlen],[array(
len,i):=maxj+1,maxj:=1,maxlen:=maxlen+1,pmax:=i,i:=i+1,j:=1],[array(a,i)<arra
y(a,pmax),j>=i,maxj+1=<maxlen],[array(len,i):=maxj+1,maxj:=1,i:=i+1,j:=1],[ar
ray(a,i)>=array(a,pmax)],[maxlen:=maxlen+1,array(len,i):=maxlen,pmax:=i,i:=i+
1,maxj:=1,j:=1])])],[]).


start :-
        open('monotone.tex',write,Stream),
        set_output(Stream),
        latex_preamble('monotone'),
        process,
        end_latex,
        close(Stream).
```

Figure 10.3: Prolog AST representation and code for starting semantic analyser on this AST

The LaTeX generated by calling `start` in the generated Prolog in Fig. 10.3 has been cut and pasted directly into this document to produce the following section[2]. We use the generated report in this next section as the basis for verifying the procedure in the subsequent section.

[2]This output has undergone minor reformatting to fit on pages the width of the thesis, however, the generated content has not been altered.

Generated output for monotone

Program 17 Code for procedure `monotone`

$\{(n > 0)\}$
procedure monotone (a : **seq Int**, n : **Int**, $maxlen$: **Int**) {

len: **seq Int**;
i, j, maxj, pmax: **Int**;
$len[1]$:= 1; $pmax$:= 1; $maxlen$:= 1; i := 2; j := 1; $maxj$:= 1 ;
do $i \leq n$ →

[len[1..i-1] are lengths of lmss ending at a[1..i-1], respectively, 1=<i=<n+1, maxlen = length of lmss in first i-1 positions, and pmax is where it ends. (When 1=<i=<n, maxj is length of lmss in a[1..j] terminating in a value less than a[i] and 1=<j=<i)]
if $a[i] < a[pmax] \wedge j < i \wedge a[(j+1)] < a[i] \wedge maxj < len[(j+1)]$ →
  $maxj$:= $len[(j+1)]$; j := $(j+1)$
[] $a[i] < a[pmax] \wedge j < i \wedge a[(j+1)] < a[i] \wedge maxj \geq len[(j+1)]$ →
  j := $(j+1)$
[] $a[i] < a[pmax] \wedge j < i \wedge a[i] \leq a[(j+1)]$ →
  j := $(j+1)$
[] $a[i] < a[pmax] \wedge j \geq i \wedge (maxj+1) > maxlen$ →
  $len[i]$:= $(maxj+1)$; $maxj$:= 1 ;
  $maxlen$:= $(maxlen+1)$; $pmax$:= i ;
  i := $(i+1)$; j := 1
[] $a[i] < a[pmax] \wedge j \geq i \wedge (maxj+1) \leq maxlen$ →
  $len[i]$:= $(maxj+1)$; $maxj$:= 1 ;
  i := $(i+1)$; j := 1
[] $a[i] \geq a[pmax]$ →
  $maxlen$:= $(maxlen+1)$; $len[i]$:= $maxlen$;
  $pmax$:= i; i := $(i+1)$;
  $maxj$:= 1; j := 1
fi
od

}

Loop Verification

For the loop

do $i \leq n$ →

[len[1..i-1] are lengths of lmss ending at a[1..i-1], respectively, 1=<i=<n+1, maxlen = length of lmss in first i-1 positions, and pmax is where it ends. (When 1=<i=<n,

```
maxj is length of lmss in a[1..j] terminating in a value
less than a[i] and 1=<j=<i)]
```

if $a[i] < a[pmax] \wedge j < i \wedge a[(j+1)] < a[i] \wedge maxj < len[(j+1)]$
$\rightarrow$

$maxj$:= $len[(j+1)]$;
j := $(j+1)$

[] $a[i] < a[pmax] \wedge j < i \wedge a[(j+1)] < a[i] \wedge maxj \geq len[(j+1)]$
$\rightarrow$

j := $(j+1)$

[] $a[i] < a[pmax] \wedge j < i \wedge a[i] \leq a[(j+1)] \rightarrow$

j := $(j+1)$

[] $a[i] < a[pmax] \wedge j \geq i \wedge (maxj+1) > maxlen \rightarrow$

$len[i]$:= $(maxj+1)$;
$maxj$:= 1 ;
$maxlen$:= $(maxlen+1)$;
$pmax$:= i ;
i := $(i+1)$;
j := 1

[] $a[i] < a[pmax] \wedge j \geq i \wedge (maxj+1) \leq maxlen \rightarrow$

$len[i]$:= $(maxj+1)$;
$maxj$:= 1 ;
i := $(i+1)$;
j := 1

[] $a[i] \geq a[pmax] \rightarrow$

$maxlen$:= $(maxlen+1)$;
$len[i]$:= $maxlen$;
$pmax$:= i ;
i := $(i+1)$;
$maxj$:= 1 ;
j := 1

fi

od

We calculate the iterative-form invariant

$$R_{k+1} \equiv \left(\begin{array}{l} \left(\begin{array}{l} a = a_k \wedge i = i_k \wedge len = len_k \wedge maxlen = maxlen_k \wedge n = n_k \wedge pmax = pmax_k \wedge \\ i \leq n \wedge k \in \mathbf{N} \wedge a[i] < a[pmax] \wedge j_k < i \wedge a[(j_k+1)] < a[i] \wedge \\ maxj_k < len[(j_k+1)] \wedge maxj = len[(j_k+1)] \wedge j = (1+j_k) \end{array} \right) \vee \\ \left(\begin{array}{l} a = a_k \wedge i = i_k \wedge len = len_k \wedge maxj = maxj_k \wedge maxlen = maxlen_k \wedge n = n_k \wedge \\ pmax = pmax_k \wedge i \leq n \wedge k \in \mathbf{N} \wedge a[i] < a[pmax] \wedge j_k < i \wedge \\ a[(j_k+1)] < a[i] \wedge maxj \geq len[(j_k+1)] \wedge j = (1+j_k) \end{array} \right) \vee \\ \left(\begin{array}{l} a = a_k \wedge i = i_k \wedge len = len_k \wedge maxj = maxj_k \wedge maxlen = maxlen_k \wedge n = n_k \wedge \\ pmax = pmax_k \wedge i \leq n \wedge k \in \mathbf{N} \wedge a[i] < a[pmax] \wedge j_k < i \wedge \\ a[i] \leq a[(j_k+1)] \wedge j = (1+j_k) \end{array} \right) \vee \\ \left(\begin{array}{l} a = a_k \wedge n = n_k \wedge i_k \leq n \wedge k \in \mathbf{N} \wedge a[i_k] < a[pmax_k] \wedge \\ j_k \geq i_k \wedge (1+maxj_k) > maxlen_k \wedge len[i_k] = (maxj_k+1) \wedge maxj = 1 \wedge \\ maxlen = (1+maxlen_k) \wedge pmax = i_k \wedge i = (1+i_k) \wedge j = 1 \end{array} \right) \vee \\ \left(\begin{array}{l} a = a_k \wedge maxlen = maxlen_k \wedge n = n_k \wedge pmax = pmax_k \wedge i_k \leq n \wedge \\ k \in \mathbf{N} \wedge a[i_k] < a[pmax] \wedge j_k \geq i_k \wedge (1+maxj_k) \leq maxlen \wedge \\ len[i_k] = (maxj_k+1) \wedge maxj = 1 \wedge i = (1+i_k) \wedge j = 1 \end{array} \right) \vee \\ \left(\begin{array}{l} a = a_k \wedge n = n_k \wedge i_k \leq n \wedge k \in \mathbf{N} \wedge a[i_k] \geq a[pmax_k] \wedge \\ maxlen = (1+maxlen_k) \wedge len[i_k] = maxlen \wedge \\ pmax = i_k \wedge i = (1+i_k) \wedge maxj = 1 \wedge j = 1 \end{array} \right) \end{array} \right)$$

where for any variable v, v_{k+1} is represented by v for brevity.

Function Theoretic Verification Obligation

Given the intended function:

```
f=[len[1..i-1] are lengths of lmss ending at a[1..i-1],
respectively, 1=<i=<n+1, maxlen = length of lmss in first i-1
positions, and pmax is where it ends.  (When 1=<i=<n, maxj is
length of lmss in a[1..j] terminating in a value less than
a[i] and 1=<j=<i)]
```

Answer the following questions:

- Does $\left(\begin{array}{l} n > 0 \wedge a = a? \wedge a \in \textbf{seq Int} \wedge n = n? \wedge n \in \textbf{Int} \wedge \\ maxlen \in \textbf{Int} \wedge len \in \textbf{seq Int} \wedge i \in \textbf{Int} \wedge j \in \textbf{Int} \wedge \\ maxj \in \textbf{Int} \wedge pmax \in \textbf{Int} \wedge len\,[1] = 1 \wedge pmax = 1 \wedge \\ maxlen = 1 \wedge i = 2 \wedge j = 1 \end{array}\right)$ establish f?
- Does f followed by R_{k+1} establish f?

Loop Progress Conditions

The loop will progress in an iteration, when the following holds at commencement of that iteration:

$$\left(\begin{array}{l} (i < (1+n) \wedge i \leq n \wedge a\,[i] < a\,[pmax] \wedge j \geq i \wedge (1 + maxj) > maxlen) \vee \\ (i < (1+n) \wedge i \leq n \wedge a\,[i] < a\,[pmax] \wedge j \geq i \wedge (1 + maxj) \leq maxlen) \vee \\ (i < (1+n) \wedge i \leq n \wedge a\,[i] \geq a\,[pmax]) \end{array}\right)$$

Invariant and Postcondition

The loop invariant and strongest postcondition could not be derived for this loop

Derived Postcondition and Verification

The following postcondition has been derived from the code in Program 17

$$[]$$

Verification Using the Report

In order to demonstrate that the loop is partially correct, we are asked to verify that the loop precondition establishes f and that f followed by R_{k+1} establishes f. That is, that f is established as an invariant. In this case f is not actually written in the style of a box specification intended function, but as a semiformal specification of invariant state on an iteration[3] . We can take a strategy similar to the inductive method of verification with an invariant of this form. We need only to slightly reword the second verification obligation to "Show that establishing R_{k+1} on an arbitrary $k+1$st iteration, knowing that f holds at the beginning of that iteration, establishes that f holds at the end of the iteration.". That is, we reword the function-theoretic proof obligation as an inductive verification step.

We begin by proving that $\begin{pmatrix} n > 0 \wedge a = a? \wedge a \in \textbf{seq Int} \wedge n = n? \wedge n \in \textbf{Int} \wedge \\ maxlen \in \textbf{Int} \wedge len \in \textbf{seq Int} \wedge i \in \textbf{Int} \wedge j \in \textbf{Int} \wedge \\ maxj \in \textbf{Int} \wedge pmax \in \textbf{Int} \wedge len\,[1] = 1 \wedge pmax = 1 \wedge \\ maxlen = 1 \wedge i = 2 \wedge j = 1 \wedge maxj = 1 \end{pmatrix}$ establishes f.

The proof of the first obligation is as follows:

Proof. We know that $i = 2 \wedge len[1] = 1$ from the calculated loop precondition, establishing `len[1..i-1] are lengths of lmss ending at a[1..i-1]`. We also know that $n > 0$ holds and so can infer that $1 \leq i \leq n+1$ also holds. The length of the longest monotone sub-sequence in the first position $(i-1 = 1)$ is by definition 1 and it ends at $pmax = 1$. The calculated loop precondition implies that these relationships hold. Additionally, the loop precondition implies that $j = 1 \wedge maxj = 1$, establishing that $maxj$ is length of longest monotone subsequence in a[1..j], and $1 \leq j \leq i$ independent of any conditions. Therefore f is established by the loop initialisation yielding the precondition given above. $\square$

To prove that the loop body maintains f on any iteration, we assume that f holds initially at the beginning of an arbitrary $k+1$st iteration and write:

> f_k=[$len\,[1..i_k - 1]_k$ `are lengths of lmss ending at` $a\,[1..i_k - 1]$`, respectively,` $1 \leq i_k \leq n+1$`,` $maxlen_k =$ `length of lmss in first` $i_k - 1$ `positions, and` $pmax_k$ `is where it ends. (When` $1 \leq i_k \leq n$`,` $maxj_k$ `is length of lmss in` $a\,[1..j_k]$ `terminating in a value less than` $a\,[i_k]$ `and` $1 \leq j_k \leq i_k$ `)]`

We then show inductively that $f_k \wedge R_{k+1} \Rightarrow f_{k+1}$, where

[3]The prototype, recognising that this is not a first order logic invariant, thinks it is an intended function and so produces a function-theoretic verification question. This example actually emphasises that the verification strategy for both types of invariants involves induction.

f_{k+1}`=[`$len\,[1..i_{k+1}-1]_{k+1}$ `are lengths of lmss ending at` $a\,[1..i_{k+1}-1]$`, respectively,` $1 \leq i_{k+1} \leq n+1$`,` $maxlen_{k+1} =$ `length of lmss in first` $i_{k+1}-1$ `positions, and` $pmax_{k+1}$ `is where it ends. (When` $1 \leq i_{k+1} \leq n$`,` $maxj_{k+1}$ `is length of lmss in` $a\,[1..j_{k+1}]$ `terminating in a value less than` $a\,[i_{k+1}]$ `and` $1 \leq j_{k+1} \leq i_{k+1}$`)]`

Proof. To prove that f is maintained as an invariant, we show that each disjunct of R_{k+1} establishes f_{k+1} given f_k. We treat each of these sub-proofs as separate cases.

Case 1 - $f_k \wedge R_{1,k+1} \Rightarrow f_{k+1}$**:**

We prove that given f_k,

$$\left(\begin{array}{l} a = a_k \wedge i = i_k \wedge len = len_k \wedge maxlen = maxlen_k \wedge n = n_k \wedge pmax = pmax_k \wedge \\ i \leq n \wedge k \in \mathbf{N} \wedge a\,[i] < a\,[pmax] \wedge j_k < i \wedge a\,[(j_k+1)] < a\,[i] \wedge \\ maxj_k < len\,[(j_k+1)] \wedge maxj = len\,[(j_k+1)] \wedge j = (1+j_k) \end{array}\right)$$

establishes f_{k+1}.

At the beginning of a $k+1$st iteration we know that $1 \leq i_k \leq n$ and that $maxj_k$ is the length of the longest monotone sub-sequence in $a\,[1..j_k]$ terminating in a value less than $a\,[i_k]$ (by f_k). $R_{1,k+1}$ implies that $len\,[j_k+1]$ is greater than $maxj_k$ at the start of the iteration, indicating that there is a monotone sub-sequence in $a\,[1..j_k+1]$ longer than $maxj$ at the start of the iteration, and that this is of length $len\,[j_k+1]$. $R_{1,k+1}$ implies that after the $k+1$st iteration $maxj$ is equal to $len\,[j_k+1]$, the length of the longest monotone sub-sequence in $a\,[1..j_k+1]$. Now $j_{k+1} = j_k+1$, so $R_{1,k+1}$ implies that $maxj$ is equal to the length of the longest monotone sub-sequence in $a\,[1..j_{k+1}]$. $R_{1,k+1}$ also implies that $a\,[j_k+1] < a\,[i_k]$ and $i_{k+1} = i_k$, so after the $k+1$st iteration $maxj_{k+1}$ is equal to the length of the longest monotone sub-sequence in $a\,[1..j_{k+1}]$ terminating in a value less than $a\,[i_{k+1}]$. We can also see that $R_{1,k+1} \Rightarrow j_k < i \wedge j_{k+1} = j_k+1$, so $j_{k+1} \leq i$ must hold. The given invariant also states that $1 \leq j_k$ so $1 < j_{k+1}$ must also hold.

We reason that if $len\,[1..i_k]_k$ are lengths of longest monotone sub-sequences ending at $a\,[1..i_k]$, respectively, holds at the beginning of the $k+1$st iteration, then because $R_{1,k+1}$ implies that i, len, $maxlen$, $pmax$ and a are unchanged, $len\,[1..i_{k+1}]_{k+1}$ are lengths of longest monotone sub-sequences ending at $a\,[1..i_{k+1}]$, respectively. Similarly, we reason that $1 \leq i_{k+1} \leq n+1$ and that $maxlen_{k+1}$ is the length of the longest monotone sub-sequence in the first $i_{k+1}-1$ positions and $pmax_{k+1}$ is where it ends.

Therefore, the path through the loop body yielding $R_{1,k+1}$ maintains the given invariant.

Case 2 - $f_k \wedge R_{2,k+1} \Rightarrow f_{k+1}$**:**

We prove that given f_k,

$$\left(\begin{array}{l} a = a_k \wedge i = i_k \wedge len = len_k \wedge maxj = maxj_k \wedge maxlen = maxlen_k \wedge n = n_k \wedge \\ pmax = pmax_k \wedge i \leq n \wedge k \in \mathbf{N} \wedge a[i] < a[pmax] \wedge j_k < i \wedge \\ a[(j_k+1)] < a[i] \wedge maxj \geq len[(j_k+1)] \wedge j = (1+j_k) \end{array}\right)$$

establishes f_{k+1}.

At the beginning of a $k+1$st iteration we know that $1 \leq i_k \leq n$ and that $maxj_k$ is the length of the longest monotone sub-sequence in $a[1..j_k]$ terminating in a value less than $a[i_k]$ (by f_k). $R_{2,k+1}$ implies that $len[j_k+1]$ is less than or equal to $maxj_{k+1}$ and that $maxj_{k+1} = maxj_k$, indicating that the monotone sub-sequence terminating at $a[j_k+1]$ is not longer than $maxj$. Now $j_{k+1} = j_k+1$, so $R_{2,k+1}$ implies that $maxj_{k+1}$ is equal to the length of the longest monotone sub-sequence in $a[1..j_{k+1}]$. $R_{2,k+1}$ also implies that $a[j_k+1] < a[i_{k+1}]$, so after the $k+1$st iteration $maxj_{k+1}$ is equal to the length of the longest monotone sub-sequence in $a[1..j_{k+1}]$ terminating in a value less than $a[i_{k+1}]$. We can also see that $R_{2,k+1} \Rightarrow j_k < i_{k+1} \wedge j_{k+1} = j_k+1$, so $j_{k+1} \leq i_{k+1}$ must hold. The given invariant also states that $1 \leq j_k$ so $1 < j_{k+1}$ must also hold.

We reason that if $len[1..i_k]_k$ are lengths of longest monotone sub-sequences ending at $a[1..i_k]$, respectively, holds at the beginning of the $k+1$st iteration, then because $R_{2,k+1}$ implies that i, len, $maxlen$, $pmax$ and a are unchanged, $len[1..i_{k+1}]_{k+1}$ are lengths of longest monotone sub-sequences ending at $a[1..i_{k+1}]$, respectively. Similarly, we reason that $1 \leq i_{k+1} \leq n+1$ and that $maxlen_{k+1}$ is the length of the longest monotone sub-sequence in the first $i_{k+1}-1$ positions and $pmax_{k+1}$ is where it ends.

Therefore, the path through the loop body yielding $R_{2,k+1}$ maintains the given invariant.

Case 3 - $f_k \wedge R_{3,k+1} \Rightarrow f_{k+1}$**:**

We prove that given f_k,

$$\left(\begin{array}{l} a = a_k \wedge i = i_k \wedge len = len_k \wedge maxj = maxj_k \wedge maxlen = maxlen_k \wedge n = n_k \wedge \\ pmax = pmax_k \wedge i \leq n \wedge k \in \mathbf{N} \wedge a[i] < a[pmax] \wedge j_k < i \wedge \\ a[i] \leq a[(j_k+1)] \wedge j = (1+j_k) \end{array}\right)$$

establishes f_{k+1}.

At the beginning of a $k+1$st iteration we know that $1 \leq i_k \leq n$ and that $maxj_k$ is the length of the longest monotone sub-sequence in $a[1..j_k]$ terminating in a value less than $a[i_k]$ (by f_k). $R_{3,k+1}$ implies that $a[i_{k+1}] \leq a[(j_k+1)]$ and that $maxj_{k+1} = maxj_k \wedge j_{k+1} = j_k+1$, so we conclude that $maxj_{k+1}$ is length of the longest monotone sub-sequence in $a[1..j_{k+1}]$ terminating in a value less than $a[i_{k+1}]$.

We also see that $R_{3,k+1} \Rightarrow j_k < i \wedge j_{k+1} = j_k+1$, so $j_{k+1} \leq i_{k+1}$ must hold. The given invariant also states that $1 \leq j_k$ so $1 < j_{k+1}$ must also hold.

We reason that if $len\,[1..i_k]_k$ are lengths of longest monotone sub-sequences ending at $a\,[1..i_k]$, respectively, holds at the beginning of the $k+1$st iteration, then because $R_{3,k+1}$ implies that i, len, $maxlen$, $pmax$ and a are unchanged, $len\,[1..i_{k+1}]_{k+1}$ are lengths of longest monotone sub-sequences ending at $a\,[1..i_{k+1}]$, respectively. Similarly, we reason that $1 \leq i_{k+1} \leq n+1$ and that $maxlen_{k+1}$ is the length of the longest monotone sub-sequence in the first $i_{k+1}-1$ positions and $pmax_{k+1}$ is where it ends.

Therefore, the path through the loop body yielding $R_{3,k+1}$ maintains the given invariant.

Case 4 - $f_k \wedge R_{4,k+1} \Rightarrow f_{k+1}$**:**

We prove that given f_k,

$$\left(\begin{array}{l} a = a_k \wedge n = n_k \wedge i_k \leq n \wedge k \in \mathbf{N} \wedge a\,[i_k] < a\,[pmax_k] \wedge \\ j_k \geq i_k \wedge (1 + maxj_k) > maxlen_k \wedge len\,[i_k] = (maxj_k + 1) \wedge maxj = 1 \wedge \\ maxlen = (1 + maxlen_k) \wedge pmax = i_k \wedge i = (1 + i_k) \wedge j = 1 \end{array}\right)$$

establishes f_{k+1}.

We reason that because $R_{4,k+1} \Rightarrow j_{k+1} = 1 \wedge maxj_{k+1} = 1$ and we know that $i_{k+1} > 1$, then $maxj_{k+1}$ is length of the longest monotone sub-sequence in $a\,[1..j_{k+1}]$ terminating in a value less than $a\,[i_{k+1}]$ and $1 \leq j_{k+1} \leq i_{k+1}$.

From f_k we know that $maxlen_k$ is the length of the longest monotone sub-sequence in $a\,[1..i_k-1]$ and that $pmax_k$ is where it ends. From $R_{4,k+1}$ we see that $(1+maxj_k) > maxlen_k$. We know from f_k that $maxj_k$ is the length of the longest monotone sub-sequence in $a\,[1..j_k]$ with a value less than $a\,[i_k]$. $R_{4,k+1}$ also tells us that $j_k \geq i_k$ and by examining the other disjuncts we see that j_k must be equal to i_k ($j_{k+1} = j_k + 1$, whenever $j_k < i_k$). Therefore, the length of the longest monotone sub-sequence in $a\,[1..i_k]$ terminating with a value less than $a\,[i_k]$ is $maxj_k$. So, $maxj_k + 1$ must be the length of the longest monotone sub-sequence ending at $a\,[i_k]$. From f_k we know that $len\,[1..i_k-1]$ are the lengths of the longest monotone sub-sequences ending at $a\,[1..i_k-1]$, respectively, and we have $R_{4,k+1} \Rightarrow len\,[i_k] = maxj_k + 1$. Therefore, $len\,[1..i_{k+1}-1]$ are the lengths of the longest monotone sub-sequences ending at $a\,[1..i_{k+1}-1]$, respectively, where $i_{k+1} = i_k + 1 (R_{4,k+1})$.

We are given that $maxlen_k$ is the length of the longest monotone sub-sequence in $a\,[1..i_k-1]$ and $pmax_k$ is the index of where it ends. We know from $R_{4,k+1}$ that $maxj_k + 1 > maxlen_k$ and that $len\,[i_k] = maxj_k + 1$ holds from above, so $maxlen_{k+1} = maxj_k + 1 \wedge pmax_{k+1} = i_k$ establishes $maxlen_{k+1}$ is the length of the longest monotone sub-sequence in the first $i_{k+1}-1$ positions $(i_{k+1} = i_k + 1)$, and $pmax_{k+1}$ is where it ends.

We are given that $1 \leq i_k \leq n+1$ and $R_{4,k+1} \Rightarrow i_{k+1} = i_k + 1 \wedge i_k \leq n$, so we conclude that $1 \leq i_{k+1} \leq n+1$ is maintained.

Therefore, the path through the loop body yielding $R_{4,k+1}$ maintains the given invariant.

Case 5 - $f_k \wedge R_{5,k+1} \Rightarrow f_{k+1}$**:**

We prove that given f_k,

$$\left(\begin{array}{l} a = a_k \wedge maxlen = maxlen_k \wedge n = n_k \wedge pmax = pmax_k \wedge i_k \leq n \wedge \\ k \in \mathbf{N} \wedge a\left[i_k\right] < a\left[pmax\right] \wedge j_k \geq i_k \wedge (1 + maxj_k) \leq maxlen \wedge \\ len\left[i_k\right] = (maxj_k + 1) \wedge maxj = 1 \wedge i = (1 + i_k) \wedge j = 1 \end{array}\right)$$

establishes f_{k+1}.

We reason that because $R_{5,k+1} \Rightarrow j_{k+1} = 1 \wedge maxj_{k+1} = 1$ and we know that $i_{k+1} > 1$, then $maxj_{k+1}$ is length of the longest monotone sub-sequence in $a\left[1..j_{k+1}\right]$ terminating in a value less than $a\left[i_{k+1}\right]$ and $1 \leq j_{k+1} \leq i_{k+1}$.

From f_k we know that $maxlen_k$ is the length of the longest monotone sub-sequence in $a\left[1..i_k - 1\right]$ and that $pmax_k$ is where it ends. From $R_{5,k+1}$ we see that $a\left[(i_k + 1)\right] < a\left[pmax_k\right]$ so does not extend this sub-sequence. Additionally, we see that $(1 + maxj_k) \leq maxlen_k$, implying that the longest monotone sub-sequence ending at $a\left[i_k\right]$ is not longer than $maxlen_k$ (see discussion in previous case regarding $maxj_k$). Now, $R_{5,k+1} \Rightarrow maxlen_{k+1} = maxlen_k \wedge pmax_{k+1} = pmax_k$, maintaining that $maxlen_{k+1}$ is the length of longest monotone sub-sequence in the first i_k positions ($i_{k+1} = i_k + 1$), and $pmax_{k+1}$ is where it ends.

From f_k we know that $len\left[1..i_k - 1\right]$ are the lengths of the longest monotone sub-sequences ending at $a\left[1..i_k - 1\right]$, respectively, and we have $R_{4,k+1} \Rightarrow len\left[i_k\right] = maxj_k + 1$, where $maxj_k$ is the length of the longest monotone sub-sequence in $a\left[1..i_k\right]$ terminating in a value less than $a\left[i_k\right]$, so $len\left[i_k\right] = maxj_k + 1$ establishes $len\left[1..i_{k+1} - 1\right]$ are the lengths of the longest monotone sub-sequences ending at $a\left[1..i_{k+1} - 1\right]$, respectively, where $i_{k+1} = i_k + 1$.

We are given that $1 \leq i_k \leq n$ and $R_{5,k+1} \Rightarrow i_{k+1} = i_k + 1 \wedge i_k \leq n$, so we conclude that $1 \leq i_{k+1} \leq n + 1$ is maintained.

Therefore, the path through the loop body yielding $R_{5,k+1}$ maintains the given invariant.

Case 6 - $f_k \wedge R_{6,k+1} \Rightarrow f_{k+1}$**:**

We prove that given f_k,

$$\left(\begin{array}{l} a = a_k \wedge n = n_k \wedge i_k \leq n \wedge k \in \mathbf{N} \wedge a\left[i_k\right] \geq a\left[pmax_k\right] \wedge \\ maxlen = (1 + maxlen_k) \wedge len\left[i_k\right] = maxlen \wedge \\ pmax = i_k \wedge i = (1 + i_k) \wedge maxj = 1 \wedge j = 1 \end{array}\right)$$

establishes f_{k+1}.

We reason that because $R_{6,k+1} \Rightarrow j_{k+1} = 1 \wedge maxj_{k+1} = 1$ and we know that $i_{k+1} > 1$, then $maxj_{k+1}$ is length of the longest monotone sub-sequence in $a\left[1..j_{k+1}\right]$ terminating in a value less than $a\left[i_{k+1}\right]$ and $1 \leq j_{k+1} \leq i_{k+1}$.

We are given that $len\left[1..i_k - 1\right]_k$ are lengths of the longest monotone sub-sequences ending at $a\left[1..i_k - 1\right]$, respectively, $1 \leq i_k \leq n + 1$, $maxlen_k$ is the length of the longest monotone sub-sequence in the first $i_k - 1$ positions, and $pmax_k$ is where it ends. $R_{6,k+1}$ implies that $a[i_k] > a[pmax_k]$ so the longest monotone sub-sequence to the first $i_{k+1} - 1$ positions ($i_{k+1} = i_k + 1$) must end at $a[i_k]$. Therefore, we expect $pmax_{k+1} = i_k \wedge maxlen_{k+1} = maxlen_k + 1 \wedge len[i_k] = maxlen_{k+1}$. We see that $R_{6,k+1}$ implies that this is established in this case. We also reason that $1 \leq i_{k+1} \leq n + 1$ is maintained in a similar manner to that in the previous two cases.

Now we find a defect. $R_{6,k+1}$ also establishes $pmax_{k+1} = i_k \wedge maxlen_{k+1} = maxlen_k + 1 \wedge len[i_k] = maxlen_{k+1}$ when $a[i_k] = a[pmax_k]$. That is, the length of the longest monotone sub-sequence is extended although, our original definition of a monotone sub-sequence as being one in which all of the elements are *strictly increasing* from left to right, is not upheld. On careful examination of the original documentation for this program in [25] we see that this procedure is only intended to find the longest monotone sub-sequence for arrays in which every element in $a[1..n]$ is unique. That is, appealing to the principles of design by contract $\forall p, q \in [1..n] \, (p \neq q \Rightarrow a[p] \neq a[q])$ is a precondition to this procedure. Under this precondition, the case where $a[i_k] = a[pmax_k]$ can not occur, therefore, $R_{6,k+1}$ satisfies the given invariant.

With this case we have identified a defect, not with the translation of the procedure, but with the precondition specified for the procedure. To correct this defect, we need to strengthen the existing precondition to

$$(n > 0 \wedge \forall p, q \in [1..n] \, (p \neq q \Rightarrow a[p] \neq a[q]))$$

Under this new precondition, we have reasoned that all paths through the loop body, and therefore the loop itself, maintain the given invariant and that the loop precondition establishes the invariant. Given that the invariant used is a suitable invariant for the loop, then we have reasoned that the loop is partially correct with respect to the specification given in [25].

Checking Loop Progress

It is also possible to reason that the loop always progresses toward termination. Looking at the derived loop progress conditions we see that the loop only progresses toward termination in an iteration when

$$\left(\begin{array}{l} (i < (1+n) \wedge i \leq n \wedge a[i] < a[pmax] \wedge j \geq i \wedge (1 + maxj) > maxlen) \vee \\ (i < (1+n) \wedge i \leq n \wedge a[i] < a[pmax] \wedge j \geq i \wedge (1 + maxj) \leq maxlen) \vee \\ (i < (1+n) \wedge i \leq n \wedge a[i] \geq a[pmax]) \end{array}\right)$$

We note that these disjuncts correspond to the last three disjuncts in R_{k+1} which in turn correspond to the three paths through the loop body in which i is increased. In the remaining disjuncts we see that the loop guard variable i is unchanged. We note that the loop progress conditions describe conditions for progress with every iteration and attempt to reason that the loop will always eventually progress toward an iteration where i is increased. We note that the three disjuncts not incrementing i involve the increment of a variable j and that these disjuncts only hold when $j < i$ at commencement of an iteration. Repeated iteration through a path yielding a disjunct in which $i_{k+1} = i_k$ is maintained, eventually results in $j_{k+1} \geq i_k$, so on a subsequent iteration we can only satisfy a disjunct which involves increment of i and resetting of j. Therefore, i is incremented infinitely often until termination. That is, the loop will always eventually progress toward termination. $\square$

10.2 Conclusion

In this chapter we have described at a high level, the architecture of a prototype tool to implement some of the algorithms and heuristics presented in this dissertation. A discussion and table have been presented describing the current extent of this implementation. We have presented a concluding example illustrating how the output generated by the tool can be used to support human verification of a non-trivial procedure. This example emphasises one of the main advantages of the method employed in this dissertation over theorem proving approaches. That being, that proof obligations to support verification can be generated without requiring a formal specification, such as is the case with theorem proving approaches to verification.

Chapter 11

Conclusions

This thesis has investigated problems related to the derivation of behavioral specifications (semantics) from program code and the use of these derived semantics to support software engineering activities. The derivation of semantics from program code has applications primarily in program verification, as well as maintenance and reuse activities. This investigation has resulted in the definition of semantic derivation techniques and methods, amenable to automation. The automation of some of the techniques developed has been proved viable in a prototype tool (discussed in the previous chapter). The methods presented in this dissertation manifest in more repeatable, effective and practical methods of inspection and formal verification.

We conclude this dissertation by re-iterating and summarising the major contributions of the research, providing a critical evaluation of the methods and techniques presented, and describing directions for future research.

11.1 Contributions

1. The techniques developed in support of this thesis are largely based on the *Strongest Postcondition* predicate transformer (*sp*). In chapter 3 we identify problems with other formal approaches for deriving semantics to support reasoning and discuss the theoretical problems with automating existing strongest postcondition calculations for assignment. We provide a definition for *sp* to calculate the strongest postcondition for assignment, which can be mechanised and which does not rely on the calculation of inverse functions or on the introduction of new variables.

   We have also formulated strongest postcondition calculations for I/O statements and function and procedure calls (see chapter 8).

2. In order to verify iterative structures, a calculable form of the strongest postcondition predicate transformer for loops needed definition. We introduced the notion of an iterative-form loop invariant (see chapter 4) derivable from any loop, and a mechanisable technique for the computation of

this iterative-form invariant. The iterative-form invariant was originally proposed as the basis for calculating conjunctive invariants, however, its usefulness in assisting inspection of certain classes of programs and for supporting verification by mathematical proof by induction and function-theoretic verification (see chapter 9), were realised.

3. We extended the algorithmic techniques of [62] for generating invariants from loops with non-branching loop bodies, concentrating on solvable difference equations in the iterative-form invariant and systematic approaches to deriving array loop invariants (chapter 6). The techniques presented in this dissertation for non-branching loops are shown to be mechanisable in a prototype tool.

4. In addition we have extended algorithmic and heuristic techniques [62, 80] for the generation of conjunctive invariants from loops with multiply-branching loop bodies by applying heuristic algorithms to transform the iterative-form invariant (chapter 6). We have based our techniques on a normalised form for loops [28] which have all accessible, logical redundancy removed from their structure. These, mostly heuristic, techniques are only capable of dealing with certain classes of loop, described in detail in chapter 6. We argue that the techniques can handle most searching, sorting, partitioning and copying algorithms. Further work needs to be conducted to extend these heuristics for other classes of problems and to continue to investigate a, possibly general, algorithmic solution to the problem.

5. We have defined a mechanisable technique for deriving loop progress and termination conditions (see chapter 7) directly from the code of the loop, as well as a process using the results to support verification and defect correction. This approach is not reliant on the provision of a formal specification or the creation of test cases, and is constructive in that the results are useful for defect correction and reuse.

6. We have described verification methods based on the derived semantic forms (see chapter 9). These techniques are based on the derived semantics and not the original program and language itself. We conjecture that because the need to provide inference rules for language constructs has been removed, these techniques may be more amenable to automated and semi-automated theorem proving than traditional approaches.

7. A prototype tool to implement the rules of chapters 4 and 5 has been developed. This prototype is discussed in detail in the next section.

11.2 Critical Evaluation of the Methods

Although the methods and techniques discussed in this thesis appear to be of significant benefit to supporting verification based inspection activities, the question of their scalability to larger programs is yet to be answered.

We have demonstrated that the algorithmic approach to the derivation of invariants presented in chapter 5 is not applicable to loops with multiple paths through the loop body. We have presented heuristic methods for deriving non-iterative invariants for certain classes of multiply-branching loops. As the methods are based on heuristics they are only known to be applicable to the classes of loops which have been examined. They are therefore, not generally applicable.

It is anticipated that the implementation of some of the algorithms in this thesis will be problematic as a result of the fact that a number of the simplifications and solutions required to be performed by the tool are complex. Specific examples include:

- Automating the manipulation of the index sets when dealing with algorithmically deriving array semantics (see chapter 5).
- Automating the complex pattern matching requirements applying the heuristics of chapter 6.
- The automation of theorem 7.4 for finding solutions to identify termination conditions using the difference equation solutions.

11.3 Future Work

The techniques and tools developed for semantic derivation and the methods discussed for their use are incomplete in a number of respects. We discuss these deficiencies with a view to describing opportunities for future work:

- The methods presented in chapters 5 and 6 for producing conjunctive invariants from an iterative-form invariant are restricted to special classes of loops (discussed in chapter 6.). In order for these techniques to be useful for supporting direct proof and providing documentation for supporting reuse in general, these methods need to be extended to deal with loops which do not represent the classes of problem discussed. Such extensions are likely to be in the form of heuristic rules based on recognisable patterns in the iterative-form invariant which indicate a certain class of program.
- The complete implementation of the computational models developed is another major challenge for the future. In this regard, the prototype tool needs to be extended to implement the heuristic algorithms of chapter 6. This is a non-trivial task as these algorithms are reliant on complex pattern matching.
- It is envisaged that the next version of the tool will deal with a real language. In order to deal with languages like C, one of the most important additions to the techniques presented would be rules of calculation to determine the semantics of pointers and pointer arithmetic. Ernst [30] states the inability of static invariant generation techniques to deal with pointers

as a major drawback of these techniques. In [39] Gannod and Cheng address the issue of deriving relational semantics from expressions involving pointers. Their treatment focusses on semantics and not alias detection, considered undecidable in static analysis [66]. Initial work on implementing the rules defined in [39] for calculating the strongest postcondition with expressions involving pointers has commenced.

- The extension of the methods, techniques and tools to be applicable to Object Oriented languages also needs consideration. Such future work must focus on the semantics of objects and object oriented systems.

- The integration of the approaches to verification discussed in chapter 9 with existing theorem-prover technology seems a plausible approach to achieving highly automated program proof of correctness. This is due to the fact that the method presented is not reliant on program language specific inference rules being added to the theorem prover. Investigation and practical work with theorem provers using the results derived from the methods in this dissertation warrants further research.

- Empirical evaluation of the actual benefits obtained by applying these methods needs to be performed. It is envisaged that this will involve the collection of data in an industry setting. Tool support for real languages will be necessary prior to such investigation.

In conclusion, powerful, comprehensive and largely automated approaches to the derivation of program semantics, and the use of those semantics to support software engineering activities, have been developed in this dissertation. We have defended the thesis presented in chapter 1:

1. Current methods of verification and defect detection are, generally, not repeatable.

2. The derivation of semantic information, including invariants, from program code can support and improve the repeatability of verification and inspection tasks.

3. Algorithms for the derivation of certain semantic representations from program code exist and can be mechanised.

4. These semantic derivation techniques are complete in the sense that certain semantic forms can always be derived directly from the program code.

This research, and future research based on it, makes an important contribution toward practical and repeatable, formal and semi-formal verification. The automation of techniques in tools to support practical, formal code reading and verification will assist in addressing the needs of trusted component technologies and the general requirement for quality in software.

Appendix A

Prototype Language Syntax

We present a BNF description of the syntax of the language used in this dissertation. This syntax definition is the same form as the grammar used by the prototype tool.

<program> ::= <block_list>

<block_list> ::= <block_list><block_stmt> | <block_stmt>

<block_stmt> ::= <procedure_stmt>|<code_block>

<procedure_stmt> ::= **procedure** <ident><param_list><code_block>

<param_list> ::= **()** | **(**<dec_list>**)**

<dec_list> ::= <qual><dec>**,**<dec_list> | <qual><dec>

<qual> ::= **IN** | **OUT** | **INOUT**

<dec> ::= <ident>**:**<type>

<code_block> ::= **{**<stmt_list>**}**

<stmt_list> ::= <statement>**;**<stmt_list> | <statement>

<statement> ::= <dec> | <const_dec> | <if_stmt> | <do_stmt> | <assign_stmt> | <read_stmt> | <write_stmt> | <procedure_call>

<const_dec> ::= <const_ident> **:=** <exp>

<if_stmt> ::= **if** <exp> → <stmt_list><more_if>

<more_if> ::= [] <exp> → <stmt_list><more_if> | **fi**

<do_stmt> ::= **do** <exp> → <stmt_list> **od**

<assign_stmt> ::= <ident> **:=** <exp>

<read_stmt> ::= **read** <ident>

<write_stmt> ::= **write** <exp>

<procedure_call> ::= <ident><actual_params>

<actual_params> ::= **()** | **(** <exp_list> **)**

<exp_list> ::= <exp>**,**<exp_list> | <exp>

<exp> ::= *An arithmetic or boolean expression using Pascal style operators*

The following are presented using EBNF:

<uc_letter> ::=[A-Z]

<lc_letter> ::= [a-z]

<letter> ::= <uc_letter> | <lc_letter>

<digit> ::= [0-9]

<ident> ::= <identifier> | <identifier><index>

<identifier> ::= <lc_letter>(<letter> | <digit>)*

<const_ident> ::= <uc_letter>(<letter> | <digit>)*

<index> ::= [<exp>]

<type> ::= $\mathbb{Z}$ | $\mathbb{N}$ | $\mathbb{R}$ | **TEXT** | **seq** <type>

Appendix B

Sample Prototype Tool Output

This remainder of this appendix has been generated entirely by the prototype tool[1]. Most of these examples correspond to examples that have been used in the dissertation and have been used to support the production of the dissertation.

B.1 qr

Program 18 Code for procedure **qr**

```
{(n > 0 ∧ d > 0)}
procedure qr (n : Int, d : Int ) {

    r := n ;
    q := 0 ;
    do r > d →
        [(n? = (q × d? + r))]
        r := (r − d) ;
        q := (q + d)
    od

}
```

B.1.1 Loop Verification

For the loop

```
do r > d →

    [(n? = (q × d? + r))]
    r := (r − d) ;
    q := (q + d)
```

[1]Some minor reformatting for long formulas has taken place.

```
od
```

We calculate the iterative-form invariant

$$R_{k+1} \equiv (d = d_k \wedge n = n_k \wedge r_k > d \wedge k \in \mathbf{N} \wedge r = (r_k - d) \wedge q = (q_k + d))$$

where for any variable v, v_{k+1} is represented by v for brevity

Inductive proof of Invariance

To prove that the invariant $(n? = (q \times d? + r))$ is established, prove the following:

Base Case:

$$\left[\begin{array}{l} (n > 0 \wedge d > 0 \wedge n = n? \wedge n \in \mathbf{Int} \wedge d = d? \wedge d \in \mathbf{Int} \wedge r = n \wedge q = 0) \Rightarrow \\ (n? = (q \times d? + r)) \end{array}\right]$$

Inductive Case:

$$\left[\begin{array}{l} (n_k? = (q_k \times d_k? + r_k)) \wedge \\ (d = d_k \wedge n = n_k \wedge r_k > d \wedge k \in \mathbf{N} \wedge r = (r_k - d) \wedge q = (q_k + d)) \Rightarrow \\ (n_{k+1}? = (q_{k+1} \times d_{k+1}? + r_{k+1})) \end{array}\right]$$

Loop Progress Conditions

The loop will progress in an iteration, when the following holds at commencement of that iteration:

$$(\ (d < r \wedge 0 < d)\ )$$

Loop Termination Conditions

The loop will terminate, iff the following is satisfiable:

$$\exists k \in \mathbb{N}\,((n - d \times k) \leq d?)$$

Invariant and Postcondition

Note, the following invariant may only be useful if the difference equations in R_{k+1} are solvable.

The derived invariant is:

$$\left(\begin{array}{l} d = d? \land n = n? \land (n - d \times k) > d \land k \in \mathbf{N} \land r = ((n - d \times k) - d) \land \\ q = \left(\left(0 + \sum_{j=0}^{k} (d?)\right) + d\right) \end{array} \right)$$

The derived postcondition for this loop is:

$$\left(\begin{array}{l} (n > 0 \land d > 0 \land n = n? \land n \in \mathbf{Int} \land d = d? \land d \in \mathbf{Int} \land r = n \land q = 0 \land r \leq d) \lor \\ \left(\begin{array}{l} d = d? \land n = n? \land (n - d \times k) > d \land k \in \mathbf{N} \land r = ((n - d \times k) - d) \land \\ q = \left(\left(0 + \sum_{j=0}^{k} (d?)\right) + d\right) \land r \leq d \end{array} \right) \end{array} \right)$$

B.1.2 Derived Postcondition and Verification

The following postcondition has been derived from the code in Program 18

$$\left(\begin{array}{l} (n > 0 \land d > 0 \land n = n? \land n \in \mathbf{Int} \land d = d? \land d \in \mathbf{Int} \land r = n \land q = 0 \land r \leq d) \lor \\ \left(\begin{array}{l} d = d? \land n = n? \land (n - d \times k) > d \land k \in \mathbf{N} \land r = ((n - d \times k) - d) \land \\ q = \left(\left(0 + \sum_{j=0}^{k} (d?)\right) + d\right) \land r \leq d \end{array} \right) \end{array} \right)$$

To prove partial correctness for this procedure you must show that the derived postcondition above implies the given postcondition

$$(n? = (q \times d? + r) \land 0 \leq r \land r < d?)$$

B.2 exp2

Program 19 Code for procedure **exp2**

$\{(base > 0 \wedge exp > 0)\}$
procedure exp2 ($base$: **Int**, exp : **Int**, ans : **Int**) {

  ans := 1 ;
  b := $base$;
  e := exp ;
  do $e > 0$ $\rightarrow$
    $[(ans \times b^e = base^{exp})]$
    if e mod $2 \neq 0$ $\rightarrow$
      ans := $ans \times b$;
      e := $(e-1)$
    [] e mod $2 = 0$ $\rightarrow$
      b := $b \times b$;
      e := e div 2
    fi
  od

}

B.2.1 Loop Verification

For the loop

do $e > 0$ $\rightarrow$
  $[(ans \times b^e = base^{exp})]$
  if e mod $2 \neq 0$ $\rightarrow$
    ans := $ans \times b$;
    e := $(e-1)$
  [] e mod $2 = 0$ $\rightarrow$
    b := $b \times b$;
    e := e div 2
  fi
od

We calculate the iterative-form invariant

$$R_{k+1} \equiv \left(\begin{array}{l} \left(\begin{array}{l} b = b_k \wedge base = base_k \wedge exp = exp_k \wedge e_k > 0 \wedge k \in \mathbf{N} \wedge \\ e_k \mod 2 \neq 0 \wedge ans = ans_k \times b \wedge e = (e_k - 1) \end{array} \right) \vee \\ \left(\begin{array}{l} ans = ans_k \wedge base = base_k \wedge exp = exp_k \wedge e_k > 0 \wedge k \in \mathbf{N} \wedge \\ e_k \mod 2 = 0 \wedge b = b_k^2 \wedge e = e_k \ \mathtt{div}\ 2 \end{array} \right) \end{array} \right)$$

where for any variable v, v_{k+1} is represented by v for brevity.

Inductive proof of Invariance

To prove that the invariant $(ans \times b^e = base^{exp})$ is established, prove the following:

Base Case:

$$\left[\begin{array}{l}\left(\begin{array}{l} base > 0 \wedge exp > 0 \wedge base = base? \wedge base \in \mathbf{Int} \wedge exp = exp? \wedge \\ exp \in \mathbf{Int} \wedge ans \in \mathbf{Int} \wedge ans = 1 \wedge b = base \wedge e = exp \end{array}\right) \Rightarrow \\ (ans \times b^e = base^{exp}) \end{array}\right]$$

Inductive Case:

$$\left[\begin{array}{l} (ans_k \times b_k^{e_k} = base_k^{exp_k}) \wedge \\ \left(\begin{array}{l} \left(\begin{array}{l} b = b_k \wedge base = base_k \wedge exp = exp_k \wedge e_k > 0 \wedge k \in \mathbf{N} \wedge \\ e_k \mod 2 \neq 0 \wedge ans = ans_k \times b \wedge e = (e_k - 1) \end{array}\right) \vee \\ \left(\begin{array}{l} ans = ans_k \wedge base = base_k \wedge exp = exp_k \wedge e_k > 0 \wedge k \in \mathbf{N} \wedge \\ e_k \mod 2 = 0 \wedge b = b_k^2 \wedge e = e_k \texttt{ div } 2 \end{array}\right) \end{array}\right) \Rightarrow \\ (ans_{k+1} \times b_{k+1}^{e_{k+1}} = base_{k+1}^{exp_{k+1}}) \end{array}\right]$$

Loop Progress Conditions

The loop will progress in an iteration, when the following holds at commencement of that iteration:

$$\left(\begin{array}{l} (0 < e \wedge e \mod 2 \neq 0) \vee \\ (0 < e \wedge e \texttt{ div } 2 < e \wedge e \mod 2 = 0) \vee \\ (0 < e \wedge e \texttt{ div } 2 < e \wedge (0 - 0) < (e - e \texttt{ div } 2) \wedge e \mod 2 = 0) \end{array}\right)$$

Invariant and Postcondition

The loop invariant and strongest postcondition could not be derived for this loop

B.2.2 Derived Postcondition and Verification

The following postcondition has been derived from the code in Program 19

$$[]$$

To prove partial correctness for this procedure you must show that the derived postcondition above implies the given postcondition

$$(ans = base^{exp})$$

B.3 gcd

Program 20 Code for procedure **gcd**

```
{true}
procedure gcd (x : Int, y : Int ) {

    do x ≠ y →
        if y < x →
            x := (x − y)
        [] x < y →
            y := (y − x)
        fi
    od

}
```

B.3.1 Loop Verification

For the loop

```
do x ≠ y →
    if y < x →
        x := (x − y)
    [] x < y →
        y := (y − x)
    fi
od
```

We calculate the iterative-form invariant

$$R_{k+1} \equiv \left(\begin{array}{l} (y = y_k \wedge x_k \neq y \wedge k \in \mathbf{N} \wedge y < x_k \wedge x = (x_k - y)) \vee \\ (x = x_k \wedge x \neq y_k \wedge k \in \mathbf{N} \wedge x < y_k \wedge y = (y_k - x)) \end{array} \right)$$

where for any variable v, v_{k+1} is represented by v for brevity.

Loop Progress Conditions

The loop will progress in an iteration, when the following holds at commencement of that iteration:

$$\left(\begin{array}{l} (x < y \wedge 0 < x) \vee \\ (y < x \wedge 0 < y) \end{array} \right)$$

Invariant and Postcondition

The loop invariant and strongest postcondition could not be derived for this loop

B.3.2 Derived Postcondition and Verification

The following postcondition has been derived from the code in Program 20

$$[]$$

B.4 sum2

Program 21 Code for procedure **sum2**

```
{true}
procedure sum2 (a : seq Int, n : Int, s : Int ) {

    i := 0 ;
    s := 0 ;
    do i ≠ n →
        i := (i + 2) ;
        s := (s + a [i])
    od

}
```

B.4.1 Loop Verification

For the loop

```
do i ≠ n →
    i := (i + 2) ;
    s := (s + a [i])
od
```

We calculate the iterative-form invariant

$$R_{k+1} \equiv \left(\begin{array}{l} a = a_k \wedge n = n_k \wedge i_k \neq n \wedge k \in \mathbf{N} \wedge i = (2 + i_k) \wedge \\ s = (s_k + a\,[i]) \end{array} \right)$$

where for any variable v, v_{k+1} is represented by v for brevity

Loop Progress Conditions

The loop will progress in an iteration, when the following holds at commencement of that iteration:

$$\left(\ (i < n) \ \right)$$

Loop Termination Conditions

The loop will terminate, iff the following is satisfiable:

$$\exists k \in \mathbb{N}\,(2 \times k = n?)$$

Invariant and Postcondition

Note, the following invariant may only be useful if the difference equations in R_{k+1} are solvable.

The derived invariant is:

$$\left(\begin{array}{l} a = a? \wedge n = n? \wedge 2 \times k \neq n \wedge k \in \mathbf{N} \wedge i = 2 \times (1 + k) \wedge \\ s = \left(\left(0 + \sum_{j=0}^{(k-1)} (a\,[2 \times ({}_{j} + 1)])\right) + a\,[i]\right) \end{array}\right)$$

The derived postcondition for this loop is:

$$\left(\begin{array}{l} (a = a? \wedge a \in \textbf{seq Int} \wedge n = n? \wedge n \in \textbf{Int} \wedge s \in \textbf{Int} \wedge i = 0 \wedge s = 0 \wedge i = n) \vee \\ \left(\begin{array}{l} a = a? \wedge n = n? \wedge 2 \times k \neq n \wedge k \in \mathbf{N} \wedge i = 2 \times (1 + k) \wedge \\ s = \left(\left(0 + \sum_{j=0}^{(k-1)} (a\,[2 \times ({}_{j} + 1)])\right) + a\,[i]\right) \wedge i = n \end{array}\right) \end{array}\right)$$

B.4.2 Derived Postcondition and Verification

The following postcondition has been derived from the code in Program 21

$$\left(\begin{array}{l} (a = a? \wedge a \in \textbf{seq Int} \wedge n = n? \wedge n \in \textbf{Int} \wedge s \in \textbf{Int} \wedge i = 0 \wedge s = 0 \wedge i = n) \vee \\ \left(\begin{array}{l} a = a? \wedge n = n? \wedge 2 \times k \neq n \wedge k \in \mathbf{N} \wedge i = 2 \times (1 + k) \wedge \\ s = \left(\left(0 + \sum_{j=0}^{(k-1)} (a\,[2 \times ({}_{j} + 1)])\right) + a\,[i]\right) \wedge i = n \end{array}\right) \end{array}\right)$$

To prove partial correctness for this procedure you must show that the derived postcondition above implies the given postcondition

s=a(0)+a(2)+...+a(n)

B.5 init

Program 22 Code for procedure **init**

```
{(n < #a)}
procedure init (a : seq Int, n : Int ) {

    i := 0 ;
    do i ≤ n →
        a [i] := 0 ;
        i := (i + 1)
    od

}
```

B.5.1 Loop Verification

For the loop

```
do i ≤ n →
    a [i] := 0 ;
    i := (i + 1)
od
```

We calculate the iterative-form invariant

$$R_{k+1} \equiv (n = n_k \wedge i_k \leq n \wedge k \in \mathbf{N} \wedge a\,[i_k] = 0 \wedge i = (1 + i_k))$$

where for any variable v, v_{k+1} is represented by v for brevity

Loop Progress Conditions

The loop will progress in an iteration, when the following holds at commencement of that iteration:

$$\Big(\ (i < (1 + n) \wedge i \leq n)\ \Big)$$

Loop Termination Conditions

The loop will terminate, iff the following is satisfiable:

$$\exists k \in \mathbb{N}\,(k > n?)$$

Invariant and Postcondition

Note, the following invariant may only be useful if the difference equations in R_{k+1} *are solvable.*

The derived invariant is:

$$\left(\begin{array}{l} n = n? \land k \leq n \land k \in \mathbf{N} \land \forall_j \in [0..k] \, (a \, [_j] = 0) \land \forall_j \in ([0..(\#a..(\#a..(\#a - 1)] - [0..k]) \, (a \, [_j] = a? \, [_j]) \land \\ i = (1 + k) \end{array} \right)$$

The derived postcondition for this loop is:

$$\left(\begin{array}{l} (n < \#a \land a = a? \land a \in \textbf{seq}\, \textbf{Int} \land n = n? \land n \in \textbf{Int} \land i = 0 \land i > n) \lor \\ \left(\begin{array}{l} n = n? \land k \leq n \land k \in \mathbf{N} \land \forall_j \in [0..k] \, (a \, [_j] = 0) \land \forall_j \in ([0..(\#a - 1)] - [0..k]) \, (a \, [_j] = a? \, [_j]) \land \\ i = (1 + k) \land i > n \end{array} \right) \end{array} \right)$$

Inspection check for boundary violation:

Do any quantified formulae contain boundary violations?

- Does $a\,[0] = 0$ violate a boundary?
- Does $a\,[k] = 0$ violate a boundary?

B.5.2 Derived Postcondition and Verification

The following postcondition has been derived from the code in Program 22

$$\left(\begin{array}{l} (n < \#a \land a = a? \land a \in \textbf{seq}\, \textbf{Int} \land n = n? \land n \in \textbf{Int} \land i = 0 \land i > n) \lor \\ \left(\begin{array}{l} n = n? \land k \leq n \land k \in \mathbf{N} \land \forall_j \in [0..k] \, (a \, [_j] = 0) \land \forall_j \in ([0..(\#a - 1)] - [0..k]) \, (a \, [_j] = a? \, [_j]) \land \\ i = (1 + k) \land i > n \end{array} \right) \end{array} \right)$$

To prove partial correctness for this procedure you must show that the derived postcondition above implies the given postcondition

$$(\forall i \in [0..n] \, (a \, [i] = 0))$$

B.6 initMod7

Program 23 Code for procedure **initMod7**

$\{(n < \#a)\}$

```
procedure initMod7 (a : seq Int, n : Int ) {

    i := 0 ;
    do i ≠ n →
        a[i] := 0 ;
        i := (i + 7)
    od

}
```

B.6.1 Loop Verification

For the loop

```
do i ≠ n →
    a[i] := 0 ;
    i := (i + 7)
od
```

We calculate the iterative-form invariant

$$R_{k+1} \equiv (n = n_k \wedge i_k \neq n \wedge k \in \mathbf{N} \wedge a[i_k] = 0 \wedge i = (7 + i_k))$$

where for any variable v, v_{k+1} is represented by v for brevity

Loop Progress Conditions

The loop will progress in an iteration, when the following holds at commencement of that iteration:

$$\left(\ (i < n) \ \right)$$

Loop Termination Conditions

The loop will terminate, iff the following is satisfiable:

$$\exists k \in \mathbb{N} \, (7 \times k = n?)$$

Invariant and Postcondition

Note, the following invariant may only be useful if the difference equations in R_{k+1} *are solvable.*

The derived invariant is:

$$\left(\begin{array}{l} n = n? \wedge 7 \times k \neq n \wedge k \in \mathbf{N} \wedge \forall_j \in [0..k]\,(a\,[7 \times_j] = 0) \wedge \\ \forall_j \in ([0..(\#a - 1)] - \{_e \in [0..k] \bullet 7 \times_e\})\,(a\,[_j] = a?\,[_j]) \wedge i = 7 \times (1 + k) \end{array}\right)$$

The derived postcondition for this loop is:

$$\left(\begin{array}{l} (n < \#a \wedge a = a? \wedge a \in \mathbf{seq}\ \mathbf{Int} \wedge n = n? \wedge n \in \mathbf{Int} \wedge i = 0 \wedge i = n) \vee \\ \left(\begin{array}{l} n = n? \wedge 7 \times k \neq n \wedge k \in \mathbf{N} \wedge \forall_j \in [0..k]\,(a\,[7 \times_j] = 0) \wedge \\ \forall_j \in ([0..(\#a - 1)] - \{_e \in [0..k] \bullet 7 \times_e\})\,(a\,[_j] = a?\,[_j]) \wedge i = 7 \times (1 + k) \wedge i = n \end{array}\right) \end{array}\right)$$

Inspection check for boundary violation:

Do any quantified formulae contain boundary violations?

- Does $a\,[7 \times 0] = 0$ violate a boundary?
- Does $a\,[7 \times k] = 0$ violate a boundary?

B.6.2 Derived Postcondition and Verification

The following postcondition has been derived from the code in Program 23

$$\left(\begin{array}{l} (n < \#a \wedge a = a? \wedge a \in \mathbf{seq}\ \mathbf{Int} \wedge n = n? \wedge n \in \mathbf{Int} \wedge i = 0 \wedge i = n) \vee \\ \left(\begin{array}{l} n = n? \wedge 7 \times k \neq n \wedge k \in \mathbf{N} \wedge \forall_j \in [0..k]\,(a\,[7 \times_j] = 0) \wedge \\ \forall_j \in ([0..(\#a - 1)] - \{_e \in [0..k] \bullet 7 \times_e\})\,(a\,[_j] = a?\,[_j]) \wedge i = 7 \times (1 + k) \wedge i = n \end{array}\right) \end{array}\right)$$

To prove partial correctness for this procedure you must show that the derived postcondition above implies the given postcondition

init every 7th element to 0

B.7 sumEven

Program 24 Code for procedure **sumEven**

```
{(0 < n)}
procedure sumEven (n : Int, s : Int ) {

    i := 0 ;
    s := 0 ;
    do i ≠ n →
        i := (i + 2) ;
        s := (s + i)
    od

}
```

B.7.1 Loop Verification

For the loop

```
do i ≠ n →
    i := (i + 2) ;
    s := (s + i)
od
```

We calculate the iterative-form invariant

$$R_{k+1} \equiv (n = n_k \wedge i_k \neq n \wedge k \in \mathbf{N} \wedge i = (2 + i_k) \wedge s = (s_k + i))$$

where for any variable v, v_{k+1} is represented by v for brevity

Loop Progress Conditions

The loop will progress in an iteration, when the following holds at commencement of that iteration:

$$(\ (i < n)\ )$$

Loop Termination Conditions

The loop will terminate, iff the following is satisfiable:

$$\exists k \in \mathbb{N}\,(2 \times k = n?)$$

Invariant and Postcondition

Note, the following invariant may only be useful if the difference equations in R_{k+1} are solvable.

The derived invariant is:

$$\left(n = n? \wedge 2 \times k \neq n \wedge k \in \mathbf{N} \wedge i = 2 \times (1 + k) \wedge s = \left(\left(0 + \sum_{j=0}^{k} (2 \times_j)\right) + i\right)\right)$$

The derived postcondition for this loop is:

$$\left(\begin{array}{l} (0 < n \wedge n = n? \wedge n \in \mathbf{Int} \wedge s \in \mathbf{Int} \wedge i = 0 \wedge s = 0 \wedge i = n) \vee \\ \left(n = n? \wedge 2 \times k \neq n \wedge k \in \mathbf{N} \wedge i = 2 \times (1 + k) \wedge s = \left(\left(0 + \sum_{j=0}^{k} (2 \times_j)\right) + i\right) \wedge i = n\right) \end{array}\right)$$

B.7.2 Derived Postcondition and Verification

The following postcondition has been derived from the code in Program 24

$$\left(\begin{array}{l} (0 < n \wedge n = n? \wedge n \in \mathbf{Int} \wedge s \in \mathbf{Int} \wedge i = 0 \wedge s = 0 \wedge i = n) \vee \\ \left(n = n? \wedge 2 \times k \neq n \wedge k \in \mathbf{N} \wedge i = 2 \times (1 + k) \wedge s = \left(\left(0 + \sum_{j=0}^{k} (2 \times_j)\right) + i\right) \wedge i = n\right) \end{array}\right)$$

To prove partial correctness for this procedure you must show that the derived postcondition above implies the given postcondition

$$\left(s = \sum_{j=0}^{\frac{n}{2}} (2 \times j) \wedge i = n\right)$$

B.8 add

Program 25 Code for procedure **add**

$\{(\#v = \#w)\}$
procedure add (v : **seq Int**, w : **seq Int**, s : **seq Int**) {

i := 0 ;
do $i < \#v$ $\rightarrow$
```
[s(i) is set to v(i) + w(i) for i in
[0..length(v)-1]]
```
$s[i]$:= $(v[i] + w[i])$;
i := $(i + 1)$
od

}

B.8.1 Loop Verification

For the loop

do $i < \#v$ $\rightarrow$
```
[s(i) is set to v(i) + w(i) for i in [0..length(v)-1]]
```
$s[i]$:= $(v[i] + w[i])$;
i := $(i + 1)$

od

We calculate the iterative-form invariant

$$R_{k+1} \equiv (v = v_k \wedge w = w_k \wedge i_k < \#v \wedge k \in \mathbf{N} \wedge s[i_k] = (v[i_k] + w[i_k]) \wedge i = (1 + i_k))$$

where for any variable v, v_{k+1} is represented by v for brevity

Function Theoretic Verification Obligation

Given the intended function:

```
f=[s(i) is set to v(i) + w(i) for i in
[0..length(v)-1]]
```

Answer the following questions:

- Does $\left(\begin{array}{l} \#v = \#w \wedge v = v? \wedge v \in \textbf{seq Int} \wedge w = w? \wedge w \in \textbf{seq Int} \wedge \\ s = s? \wedge s \in \textbf{seq Int} \wedge i = 0 \end{array}\right)$ establish f?
- Does f followed by R_{k+1} establish f?

Loop Progress Conditions

The loop will progress in an iteration, when the following holds at commencement of that iteration:

$$\left(\begin{array}{l}(i < \#v) \vee \\ (i < \#v \wedge (\#v - \#v) < ((1+i) - i))\end{array}\right)$$

Loop Termination Conditions

The loop will terminate, iff the following is satisfiable:

$$\exists k \in \mathbb{N}\,(k \geq \#v?)$$

Invariant and Postcondition

Note, the following invariant may only be useful if the difference equations in R_{k+1} are solvable.

The derived invariant is:

$$\left(\begin{array}{l}v = v? \wedge w = w? \wedge k < \#v \wedge k \in \mathbf{N} \wedge \forall_j \in [0..k]\,(s\,[_j] = (v\,[_j] + w\,[_j])) \wedge \\ \forall_j \in ([0..(\#s - 1)] - [0..k])\,(s\,[_j] = s?\,[_j]) \wedge i = (1 + k)\end{array}\right)$$

The derived postcondition for this loop is:

$$\left(\begin{array}{l}(\#v = \#w \wedge v = v? \wedge v \in \textbf{seq Int} \wedge w = w? \wedge w \in \textbf{seq Int} \wedge s = s? \wedge s \in \textbf{seq Int} \wedge i = 0 \wedge i \geq \#v) \vee \\ \left(\begin{array}{l}v = v? \wedge w = w? \wedge k < \#v \wedge k \in \mathbf{N} \wedge \forall_j \in [0..k]\,(s\,[_j] = (v\,[_j] + w\,[_j])) \wedge \\ \forall_j \in ([0..(\#s - 1)] - [0..k])\,(s\,[_j] = s?\,[_j]) \wedge i = (1 + k) \wedge i \geq \#v\end{array}\right)\end{array}\right)$$

Inspection check for boundary violation:

Do any quantified formulae contain boundary violations?

- Does $s\,[0] = (v\,[0] + w\,[0])$ violate a boundary?
- Does $s\,[k] = (v\,[k] + w\,[k])$ violate a boundary?

B.8.2 Derived Postcondition and Verification

The following postcondition has been derived from the code in Program 25

$$\left(\begin{array}{l}(\#v = \#w \wedge v = v? \wedge v \in \textbf{seq Int} \wedge w = w? \wedge w \in \textbf{seq Int} \wedge s = s? \wedge s \in \textbf{seq Int} \wedge i = 0 \wedge i \geq \#v) \vee \\ \left(\begin{array}{l}v = v? \wedge w = w? \wedge k < \#v \wedge k \in \mathbf{N} \wedge \forall_j \in [0..k]\,(s\,[_j] = (v\,[_j] + w\,[_j])) \wedge \\ \forall_j \in ([0..(\#s - 1)] - [0..k])\,(s\,[_j] = s?\,[_j]) \wedge i = (1 + k) \wedge i \geq \#v\end{array}\right)\end{array}\right)$$

B.9 push

Program 26 Code for procedure **push**

$\{(top < MAXSTACK)\}$
procedure push (s : **seq Int**, top : **Int**, val : **Int**) {

$s\,[top]$:= val ;
top := $(top + 1)$

}

B.9.1 Derived Postcondition and Verification

The following postcondition has been derived from the code in Program 26

$$\left(\begin{array}{l} top? < MAXSTACK \wedge s \in \mathbf{seq\ Int} \wedge top \in \mathbf{Int} \wedge val = val? \wedge val \in \mathbf{Int} \wedge \\ s\,[top?] = val \wedge top = (top? + 1) \end{array}\right)$$

B.10 pop

Program 27 Code for procedure **pop**

```
{(top > 0)}
procedure pop (s : seq Int, top : Int ) {

    top := (top − 1)

}
```

B.10.1 Derived Postcondition and Verification

The following postcondition has been derived from the code in Program 27

$$(top? > 0 \land s = s? \land s \in \mathbf{seq\ Int} \land top \in \mathbf{Int} \land top = (top? - 1))$$

Bibliography

[1] A. Ackerman, L. Buchwald, and F. Leniski. Software inspection: An effective verification process. *IEEE Software*, 6(3):31–36, May 1989.

[2] S. Alagic and M. Arbib. *The Design of Well Strutcured and Correct Programs*. Springer-Verlag, 1978.

[3] R. Back. A calculus of refinements for program derivation. *Acta Informatica*, 25:593–625, 1988.

[4] R. Backhouse. *Program Construction and Verification*. Prentice Hall, 1986.

[5] J. Barnes. *High Integrity ADA: The Spark Approach*. Addison-Wesley, 1997.

[6] V. Basili and H. Mills. Understanding and documenting programs'. *IEEE Transactions on Software Engineering*, 8(3):270–283, May 1982.

[7] V. Basili and R. Selby. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, 13(12), December 1987.

[8] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[9] S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful techniques for the automatic generation of invariants. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 323–335, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[10] D. Bisant and J. Lysle. A two-person inspection method to improve programmer productivity. *IEEE Transactions on Software Engineering*, 15(10):1294–1304, Oct 1989.

[11] N. Bjorner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[12] B. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[13] R. Boyer and J. Moore. *A Computational Logic.* Academic Press, 1979.

[14] R. Boyer and J. Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[15] R. Boyer and J. Moore. A theorem prover for a computational logic. In M. Stickel, editor, *Proceedings 10th International Conference on Automated Deduction, Kaiserslautern (Germany)*, volume 449, pages 1–15. Springer-Verlag, 1990.

[16] R. Britcher. Using inspections to investigate program correctness. *IEEE Computer*, pages 38–44, November 1988.

[17] D. Carrington and P. Stocks. A tale of two paradigms: Formal methods and software testing. In J. Bowen and J. Hall, editors, *Z User Workshop, Cambridge 1994*, Workshops on Computing, pages 51–68. Springer-Verlag, 1994.

[18] M. Deck. Cleanroom review techniques for application development. In *Proc. 6th International Conference on Software Quality*, 1996.

[19] F. Deremer and H.Kron. Programming in the large versus programming in the small. *IEEE Transactions On Software Engineering*, 2(2), June 1976.

[20] M. Deutsch. *Software Verification and Validation: Realistic Project Approaches.* Prentice-Hall, 1982.

[21] E. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. ACM*, 18:453–457, 1975.

[22] E. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[23] E. Dijkstra and W. Feijen. *A Method of Programming.* Addison-Wesley, 1988.

[24] E. Dijkstra and C. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1989.

[25] R. Dromey. *How to Solve it by Computer.* Addison-Wesley, 1982.

[26] R. Dromey. *Program Derivation.* Addison-Wesley, 1989.

[27] R. Dromey and D. Billington. Stepwise program derivation. Technical Report SQI-91-02, Software Quality Institute, Griffith University, 1991.

[28] R. Dromey and S. Pan. Re-engineering loops. *The Computer Journal*, 39(3), 1996.

[29] M. Dyer. *The Cleanroom Approach to Quality Software Development.* Wiley, 1992.

[30] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[31] M. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[32] M. Fagan. Advances in software inspection. *IEEE Transactions on Software Engineering*, 12(7), Jul 1986.

[33] S. Flowers. *Software Failure: Management Failure*. John Wiley and Sons, 1996.

[34] R. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

[35] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[36] M. Fuchs. A data mining approach to support the creation of loop invariants using genetic programming. Technical Report TR-ARP-09-98, Centre for Information Science Research, Australian National University, 1998.

[37] G. Gannod and B. Cheng. A two-phase approach to reverse engineering using formal methods. In *Formal Methods in Programming and Their Applications*, pages 335–348, 1993.

[38] G. Gannod and B. Cheng. Strongest postcondition semantics as the formal basis for reverse enigineering. *Automated Software Engineering*, 3:139–164, 1996.

[39] G. Gannod and B. Cheng. A formal automated approach for reverse engineering programs with pointers. In *Automated Software Engineering*, pages 219–226, 1997.

[40] G. Gannod and B. C. Cheng. A formal approach for reverse engineering: A case study. In *Working Conference on Reverse Engineering*, pages 100–111, 1999.

[41] S. German and B. Wegbreit. A synthesizer of inductive assertions. *IEEE transactions on Software Engineering*, 1(1):68–75, March 1975.

[42] T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley, 1993.

[43] R. Glass. *Software Runnaways, Lessons Learned from Massive Software Project Failures*. Prentice Hall, 1998.

[44] J. Goodenough and S. Gerhart. Towards a theory of test data selection. *IEEE*, SE-1(2):156–173, 1975.

[45] M. Gordon and T. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[46] M. Gordon, R. Milner, and C. Wadsworth. Edinburgh lcf: A mechanised logic of computation. *Lecture Notes in Computer Science*, 78, 1979.

[47] D. Gries. *The Science of Programming.* Springer-Verlag, 1981.

[48] W. G. Griswold and D. Notkin. Program restructuring as an aid to software maintenance. Technical report, Seattle, WA, USA, August 1990.

[49] J. Guttag and J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[50] A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, Sep 1990.

[51] P. Hall. Relationship between specifications and testing. *Information and Software Technology*, 33(1):47–52, 1991.

[52] M. Hinchey and J. Bowen. Seven more myths of formal methods. Technical Report PGR-TR-&-94, Oxford University Computing Laboratory, 1997.

[53] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[54] C. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Lecture Notes in Computer Science*, volume 188, pages 102–116, 1971.

[55] C. Hoare. How did software get so reliable without proof. In *FME '96 Industrial Symposium of Formal Methods in Europe, 3rd Edition*, 1996.

[56] W. Howden. An evaluation of the effectiveness of symbolic testing. *Software Practice and Experience*, 8:381–397, 1978.

[57] W. Humphrey. *Managing the Software Process.* Addison-Wesley, 1989.

[58] A. Ireland and J. Stark. On the automatic discovery of loop invariants. In *Proc. of the Fourth NASA Langley Formal Methods Workshop - NASA Conference Publication 3356*, 1997.

[59] A. Jackson and D. Hoffman. Inspecting module interface specifications. *Software Testing, Verification and Reliability*, 4:101–117, 1994.

[60] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *ICSM*, pages 736–743, 2001.

[61] S. Katz and Z. Manna. A heuristic approach to program verification. In *IJCAI-73. International Joint Conference on Artificial Intelligence*, pages 500–511, 1973.

[62] S. Katz and Z. Manna. Logical analysis of programs. *Communications of the ACM*, 19(4):188–206, 1976.

[63] M. Kaufmann and J. Moore. Design goals of acl. Technical Report 101, Computational Logic, Inc., Aug. 1994.

[64] B. Kitchenham, A. Kitchenham, and J. Fellows. Effects of inspections on software quality and productivity. *ICL Technical Journal*, 5(1):112–122, May 1986.

[65] J. Knight and E. Myers. An improved inspection technique. *Communications of the ACM*, 36(11):51–61, Nov 1993.

[66] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[67] R. Linger. Cleanroom process model. *IEEE Software*, pages 50–58, Mar 1994.

[68] R. Linger and H. Mills. A case study in cleanroom software engineering: The ibm cobol structuring facility. In *Proc. 12th International Computer Science and Applications Conference*, pages 10–17. IEEE CS Press, 1988.

[69] R. Linger, H. Mills, and B. Witt. *Structured Programming: Theory and Practice*. Addison-Wesley, 1979.

[70] F. MacDonald and J. Miller. A comparison of tool-based and paper-based software inspection. *Empirical Software Engineering: An International Journal*, 3(3):233–253, September 1998.

[71] F. MacDonald, J. Miller, A. Brooks, M. Roper, and M. Wood. A review of tool support for software inspection. Technical Report RR-95-181, University of Strathclyde, Glasgow, Scotland, 1995.

[72] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

[73] P. McCarthy, A. Porter, H. Siy, and L. Votta. An experiment to assess cost-benefits of inspection meetings and their alternatives. Technical report, University of Maryland, MD, Sep 1995.

[74] B. Meyer. *Eiffel The Language*. Prentice-Hall, 1992.

[75] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[76] P. Murphy. A review of cleanroom software engineering. `citeseer.nj.nec.com/98391.html`.

[77] NASA. *Formal Methods Specification and Verification Guidebook for Software and Computer Systems, NASA-GB-002-95, Release 1.0*, 1995.

[78] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, Urbana-Champaign, IL, USA, 1992.

[79] O. Oskarsson and R. Glass. *An ISO 9000 Approach to Building Quality Software*. Prentice-Hall, 1996.

[80] S. Pan. *Software Quality Improvement, Specification, Derivation and Measurement using Formal Methods.* PhD thesis, School of Computing and Information Technology, Griffith University, 1994.

[81] D. Parnas and D. Weiss. Active design reviews: Principles and practices. *Journal of Systems and Software*, (7):259–265, 1987.

[82] L. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[83] L. Paulson. Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*, 828, 1994.

[84] A. Porter, H. Siy, A. Mockus, and L. Votta. Understanding the sources of variation in software inspections. *ACM Transactions on Software Engineering and Methodology*, 7(1):41–79, 1998.

[85] D. Powell. Deriving verification conditions and program assertions to support software inspection. In *Proc. 9th Asia Pacific Software Engineering Conference*, 2002.

[86] S. Prowell, C. Trammell, R. Linger, and J. Poore. *Cleanroom Software Engineering: Technology and Process.* Addison-Wesley Professional, 1999.

[87] J. Spivey. The z notation: a reference manual. `http://spivey.oriel.ox.ac.uk/~mike/zrm/`, 2001.

[88] J. Stark and A. Ireland. Invariant discovery via failed proof attempts. *Lecture Notes in Computer Science*, 1559:271–288, 1998.

[89] P. Stocks. *Applying Formal Methods to Software Testing.* PhD thesis, University of Queensland, St. Lucia 4072, Australia, 1993.

[90] S. Strauss and R. Ebenau. *Software Inspection Process.* McGraw-Hill, 1994.

[91] M. Utting. The ergo 5 generic proof engine. Technical Report 97-44, Software Verification Research Centre, The University of Queensland, Dec. 1997.

[92] L. Votta. Does every inspection need a meeting? In *Proceedings of ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering*, pages 107–114. ACM, Dec 1993.

[93] E. Weyuker and T. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Tansactions on Software Engineering*, 6(3), 1980.