An Investigation Into a Probabilistic Framework for Studying Symbolic Execution Based Program Conditioning

By

Mohammed Daoudi

This thesis is submitted in partial fulfilment of the requirements

for the degree of Doctor Of Philosophy

in the

Department of Computing

Goldsmiths College

University of London

New Cross, London SE14 6NW, UK.

Abstract

This thesis is concerned with program conditioning, a technique which combines symbolic execution [76, 33, 109] and theorem proving [] to identify and remove a set of statements which cannot be executed when a condition of interest holds at some point in a program. It has been applied to problems in maintenance, testing, reuse and reengineering.

All current program conditioning algorithms tend to be exponential. This is due to the fact that the computational cost of a conditioning algorithm is dominated by both the exponential growth of the conditioned state and the response time of the theorem prover.

This thesis reports on a lightweight approach to program conditioning using the FermaT simplify decision procedure. This is used as a component to *ConSUS*, a program conditioning system for a subset of the Wide Spectrum Language WSL. The thesis describes the symbolic execution algorithm used by *ConSUS*, which prunes as it conditions.

The thesis also gives analytical evidence that, although exponential in the worst case, on average, the conditioning system reduces its exponential behaviour by several orders of magnitude, thereby making it more effecient than previous approaches.

Contents

List of Figures

Chapter 1

Introduction

1.1 Aim

The aim of this thesis is to present an intraprocedural program conditioning algorithm and its implementation for a subset of the Wide Spectrum Language WSL. This algorithm/implemantation forms a central part of the conditioned slicing system *ConSUS*.

The thesis also aims to present a stochastic framework whereby it is possible to compare the approach reported here to previous approaches, model both the expected computational cost of the algorithm, and the expected tradeoff between precision and speed when constraining the time allocated to the analysis.

In this chapter, Section 1.2 briefly outlines the broad field of study and then leads into the focus of the research problem. Section 1.3 outlines the core idea of the research. Section 1.4 presents a justification for this thesis research problem. Section 1.5 outlines an introductory overview of the methodology. Section 1.6 outlines the contributions of this work. Section 1.7 briefly describes each chapter of this thesis. Section 1.8 key and controversial terms are defined. Section 1.9 delimitations of scope and key assumptions are presented. Finally, Section 1.10 summarizes this chapter.

1.2 Background to the research

Understanding a large program can be a daunting task. Program slicing [] is useful for many applications involving program comprehension [], because it reduces the size of the program, making the task less daunting. Applications which require comprehension and which use slicing include maintenance [], debugging [], and the identification and reuse of subcomponents []. All of these approaches share the observation that it is not essential to understand the behaviour and contribution of *all* of the program. Rather, it is possible, indeed preferable, to concentrate solely upon some part of the overall computation.

The original formulation of slicing was static. That is, the slicing *criterion* contained no information about the input to the program. Later work on slicing created different paradigms for slicing including dynamic slicing [] (for which the input is known) and quasi-static slicing [] (for which an input prefix is known).

Static slicing has now reached a mature stage of development. Tools, such as the Gramatech CodeSurfer system [], can efficiently slice real-worl **C** programs of the order of hundreds of thousands of lines of code in reasonable time [].

This research is concerned with a generalization of slicing called conditioned slicing. Conditioned slicing forms a theoretical bridge between the two extremes of static and dynamic slicing. It augments the traditional slicing criterion with a condition which imposes a constraint on a particular state, and/or particular variables. This additional condition can be used to simplify the program before applying a traditional static slicing algorithm. Such pre-simplification is called conditioning. and it is achieved by eliminating statements which do not contribute to the computation of the variables of interest when the program execution satisfies the condition. However, the extra simplification potential of program conditiong comes at a high computational cost.

This thesis focuses on the *conditioning* process of conditioned slicing. Program condi-

tioning involves attempting to simplify a program assuming that the states it reaches at various points in its execution satisfy certain properties. These properties are specified by adding *assertions* at arbitrary points in the program.

The construction of a conditioned program is computationally more expensive than that of a static slice. This is because program conditioning requires both symbolic execution and theorem proving. However, there exist several implementations, which are capable of producing conditioned slices for small programs.

1.3 The problem

- This section outlines the idea of the research, starting with the research problem.
- The problem addressed in this research is:
- Essentially I argue that
- or I conclude that ..... I also propose a new agenda for future research which centres on a few, key research areas and opens up research to new ....
- After the research problem and a brief summary of how it will be solved is presented, *section 1.3 presents the research questions or hypotheses*.

1.4 Justification for the research

a thesis about Program Conditioning could justify its research problem through:

- importance of program conditioning
- relative neglect of the specific research problem by previous researchers

- this section should emphasis the whole research problem
- it is not simply used for the sake of novelty
- usefulness of potential applications of the research's findings

1.5 The approach

- This section is an introductory overview of the approach.
- The section should refer to sections where the methodology is justified and described.
- justify the chosen methodology based upon the purpose of the research (wanting to optomize previous approach)

The importance of the approach advocated in this report is that the performance time is exponential in the number of control flow statementof the computed conditioned program unlike in previous approaches where this is exponential in the number of control flow statements of the program to be conditioned.

1.6 Outline of Contributions

The main contributions of this thesis are:

1.
2.
3.

1.7 Outline of the thesis

Each chapter is briefly described in this section.

1.8 Definitions

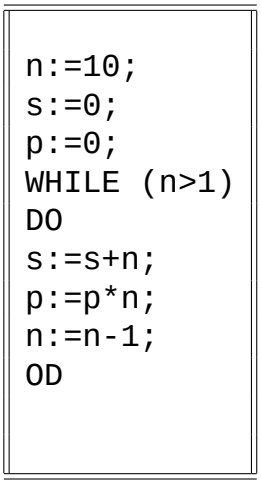
Key and controversial terms are defined in this section.

1.9 Delimitations of scope and key assumptions

Object Language, **Conditions** considered in this research.

1.10 Conclusion

This chapter laid the foundations for the report. It introduced the research problem and research questions. Then the research was justified, definitions were presented, the approach was briefly described and justified, the report was outlined, and the limitations were given. On these foundations, the report can proceed with a detailed description of the research.



```
n:=10;
s:=0;
p:=0;
WHILE (n>1)
DO
s:=s+n;
p:=p*n;
n:=n-1;
OD
```

Figure 1.1: A program fragment to be statically sliced

Program slicing is a program analysis technique that reduces a program to those statements that are relevant for a particular computation.

The original motivation for program slicing was to aid the location of faults during debugging activities. The idea was that the slice will contain the fault, but would not contain lines of code which could not have caused the failure observed. This is achieved by setting the slicing criterion to be the variable for which an incorrect value is observed.

Consider the example in Figure 1.1. The program is supposed to calculate the sum and product of the sequence of numbers from 1 to 10, but the value of the product p is always found to be zero. In order to locate the cause of this errant behaviour, we can construct a static slice for the variable p at the end of the program, as shown in Figure 1.2. Since the slice is simpler than the original program, it is easier to locate the bug (in this case, p should be initialises to 1, instead of 0).

Although static slicing can assist in simplifying programs, the slices constructed by static

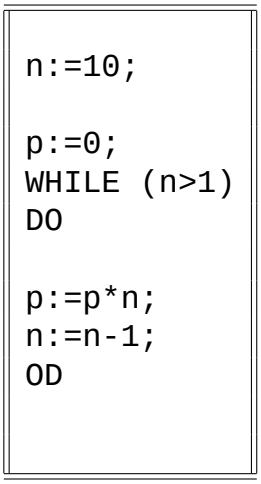


Figure 1.2: A static slice of 1.1

slicing tend to be rather large. This is particularly true for well-constructed programs, which are typically highly cohesive. This high level of cohesion results in programs where the computation of the value of each variable is highly dependent upon the values of many other variables.

The original formulation of slicing [126] was static. That is, the slicing criterion contained no information about the input to the program. Fortunately, we can provide information to the slicing tool about the input without being so specific as to give the precise values.

Consider the program in Figure 1.3, we can use a boolean expression, for example `x=y+4`, to relate the possible values of the two inputs `x` and `y`. When the program is executed in a state that satisfies this boolean condition, we know that the assignment `z:=2;` will not be executed. Any slice constructed with respect to this condition may therefore omit that statement. This approach to slicing is called *conditioned slicing*, because the slice is conditioned by knowledge about the condition in which the program is to be executed.

Conditioned slicing adresses just the kind of problems software maintainers face when presented with the task of understanding large legacy systems. Often, in this situation, we find ourselves asking questions such as:

> 'Suppose we know that `x` is greater than `y` and that `z` is equal to 4, then which statements would effect the value of the variable `v` at line 38 in the program'.

Using conditioned slicing, we can obtain an answer to this question automatically. The slice would be constructed for `v`, at 38, on `x>y AND z=4`. By building up a collage of conditioned slices which isolate different aspects of the program's behaviour, we can quickly obtain a picture of how the program behaves under various conditions. Conditioned slicing is really a tool-assisted form of the familiar approach to program comprehension of devide and conquer.

A conditioned slice can be computed by first simplifying the program with respect to the condition on the input (i.e., discarding infeasible paths with respect to the input condition) and then computing a slice on the reduced program. A symbolic executor [75, 34] can be used to compute the reduced program, also called a *conditioned program* in [18].

Crucial to conditioned slicing is the *conditioning* process. Program conditioning involves attempting to simplify a program assuming that the states it reaches at various points in its execution satisfy certain properties. These properties are specified by adding *assertions* at arbitrary points in the program. Program conditioning relies upon both *symbolic execution* and *reasoning* about symbolic predicates and therefore requires some form of automated theorem proving.

The simplifying power of a program conditioner depends on two things:

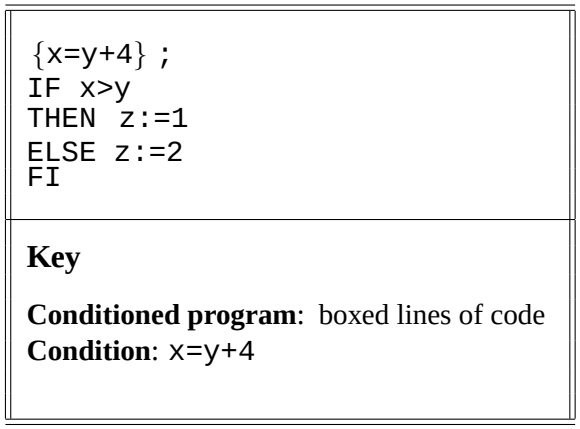


Figure 1.3: A conditioned program

1. The precision of the symbolic executor which handles propagation of state and path information.

2. The power of the underlying theorem prover which determines the truth of propositions about states and paths.

Unfortunately, implementation is not straightforward because the full exploitation of conditions requires the combination of symbolic execution and theorem proving. Hitherto, this difficultly has hindered development of fully automated conditioning slicing tools. Fox *et al.* describe the first fully automated conditioned slicing system, *ConSIT* [36]. They detaile the theory that underlies it, its architecture and the way it combines symbolic execution, theorem proving and slicing technologies .

The problem with *ConSIT*'s conditioning algorithm is that it is exponential even in the best case as described in section 2.5. *ConSIT* generates all possible paths to each statement, and have to check the accessibility of each one. One way in which this can be improved is to "fold" the reasoning and symbolic execution processes together.

In this thesis we show that we can instead make use of the monotonicity of the propositions that we have to analyse: if a path becomes infeasible, then it will remain infea-

sible for all subsequent statements. The algorithm defined in this thesis is at the heart of *ConSUS*, a light-weight program conditioned slicer for WSL. *ConSUS*' conditioner prunes symbolic execution paths based on the validity of path conditions, thereby removing unreachable code. Unlike *ConSIT*, the *ConSUS* system integrates the reasoning and symbolic execution within a single system. The symbolic executor can eliminate paths which can be determined to be unexecutable in the current symbolic state. This pruning effect makes the algorithm more *efficient* as it has a significant effect on the size of the propositions handed to the theorem prover, thus speeding up the analysis. Furthermore, the reasoning is achieved, not using the full power of a general purpose theorem prover[1], but rather by using either the in-built expression simplifier of *FermaT Simplify* or the Co-operating validity checker *CVC* in it's lightweight -*SAT* mode. This is a lightweight approach that may be capable of scaling to large programs.

We use both of these validity checkers because in some cases the performance achieved using *CVC* is better than that of *FermaT* `Simplify`. This is because the reasoning power of *CVC* does in some cases result in 'early pruning' which is missed by the less powerful validity checker *Simplify*. For example the program in Figure 1.4 is simplified when using the *ConSUS* algorithm in conjunction with *CVC* in place of *FermaT* `Simplify` which fails to remove any statements since it is unaware of the transitivity of $>$.

The main contributions of this thesis are:

1. To define a new more efficient algorithm and implementation for program conditioning which uses on-the-fly pruning of symbolic execution paths.
2. To report on empirical studies which demonstrate that
   (a) On small 'real programs' this algorithm produces a considerable reduction in program size when used with and without a program slicer.

[1] with ConSIT The test of consistency of each set of states is computed using the `Isabelle` theorem prover [97, 98, 96], as described in more detail in Section 2.5.2.

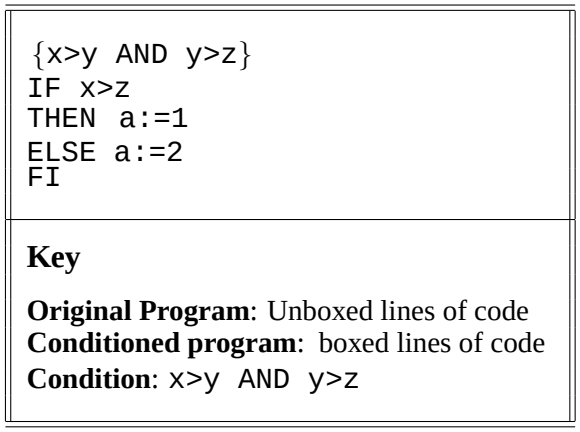


Figure 1.4: Conditioning a simple program using CVC

(b) The *ConSUS* algorithm when used in conjunction with two validity checkers: WSL's *FermaT* `Simplify` and CVC [112], has the potential for 'scaling up' for use on larger systems.

The rest of this thesis is organised as follows:

- **Chapter 2** surveys statement-deletion based slicing methods for programs written in procedural languages, and their applications. Additionally, this chapter describes previous work on symbolic execution. This includes a discussion of the ideas and motivations behind different approaches, as well as a survey of existing symbolic execution systems. Furthermore, the theoritical foundations of *FermaT*, *FermaT* `Simplify` transformation and *CVC* are reviewed. Finally this chapter presents a detailed discussion of the *ConSIT* system issues, and several ways in which it could be improved.

- **Chapter 3** describes the use of conditioned slicing to assist partition testing, illustrating this with a case study. The chapter shows how *ConSUS* can be used to provide confidence in the uniformity hypothesis for correct programs, to aid fault detection in incorrect programs and to highlight special cases.

- **chapter 4** is the main body of the thesis. It introduces an integrated approach to symbolic execution that combines reasoning and symbolic execution to prune paths as the symbolic execution proceeds.

- **Chapter 5** describes our use of both the *FermaT* `Simplify` transformation and *CVC* to achieve a form of light-weight theorem proving, which is required to determine the outcome of symbolic predicates in a symbolic conditioned-state pair.

- **Chapter ??** presents the results of an empirical investigation into the performance and scalability of the approach.

- **Chapter ??** gives a summary of our contributions and a discussion of the future direction of our work.

- Finally, the appendices contain the WSL code for ConSUS, as well as the real-world programs used in chapter **??**.

Chapter 2

Background

2.1 Program Slicing

In this section we describe different types of program slicing.

2.1.1 Static Slicing

From a formal point of view, the definition of a slice is based on the concept of slicing criterion. According to Weiser [126], a slicing criterion is a pair $(V,\ n)$ where n is a program point and V is a set of program variables. A program slice on the slicing criterion $(V,\ n)$ is a sequence of program statements that preserves the behaviour of the original program at the program point n with respect to the program variables in V, i.e. the values of the variables in V at program point n are the same in both the original program and the slice. As the behaviour of the original program has to be preserved on any input, Weiser's slicing has been called `static slicing`, to differentiate it from other forms of slicing that require the behaviour to be preserved on a subset of input to the program. This form of slice has also been defined as a `backward slice`, in contrast to a `forward slice`, defined as the set of program statements and predicates affected by the computation of the value of the variable v at a program point n [65].

Weiser has demonstrated that computing the minimal subset of statement that satisfies this requirement is undecidable [124, 126]. However, an approximation can be found by computing the least solution to a set of dataflow equations relating a Control Flow Graph (CFG) node to the variables which are relevant at that node with respect to the slicing criterion [126].

The algorithm proposed by Weiser led to an alternative definition: a slice consists of the sequence of program statements and predicates that directly or indirectly affect the computation of the variables in V before the execution of n. Building on this definition, a different algorithm has been proposed that computes slices as backwards traversals of the Program Dependence Graph (PDG) [95], a program representation where nodes represent statements and predicates, while edges carry information about control and data dependence. A slice with respect to such slicing criterion $(V,\ n)$ consists of the set of nodes that directly or indirectly affects the computation of the variables in V at node n .

Howitz `et al`. [65] extended the PDG based algorithm to compute *interprocedural* slices on the System Dependence Graph (SDG). The authors demonstrated that their algorithm is more accurate than the original interprocedral slicing algorithm by Weiser [126], because it accounts for procedure calling context. Recent improvements of algorithms to compute slices through graph reachability are presented in [100].

A parallel slicing algorithm has been presented by Danicic `et al`. [38] in which the control flow graph of a program is converted into a network of concurrent processes whose parallel execution produces the slice. Algorithms have also been proposed that compute backward static slices in the presence of arbitrary control flow [1, 6, 22, 58, 108] and pointers [87, 86].

Different applications of static slicing have been proposed in the literature, toguether with some variants on the original definition. For example, Gallagher and Lyle [51] introduced the concept of *decomposition slicing* and discussed its application to software maintenance. A decomposition slice is defined with respect to a variable v, independently of any program point n. it is given by the union of the static slices computed with respect to the variable v at all possible program points n. Other applications of program slicing include software testing [12, 53, 56, 62], program debugging [125, 88], measurement [94, 11, 92, 93], validation [82], program parallelisation [126], program integration [64], reverse engineering, comprehension [9, 41], program restructuring [19, 23, 83], and identification of reusable functions [25].

2.1.2 Dynamic Slicing

As already noted, program slicing was first proposed as a tool for decomposing programs during debugging, in order to allow a better understanding of the portion of code which revealed an error [125, 126]. In this case, the slicing criterion contains the variables which produced an unexpected results on some input of the program. However, a static slice may very often contain statements which have no influence on the values of the variables of interest for the particular execution in which the anomalous behavior of the program was discovered.

Korel and Lasky [79, 85] proposed an alternative slicing definition namely *dynamic slicing*, which uses dynamic analysis to identify all and only the statements that effect the variables of interest on the particular anomalous execution trace. In this way the size of the slice can be considerably reduced, thus allowing a better localisation of the bugs. Another advantage of dynamic slicing is the run-time handling of arrays and pointer variables. Dynamic slicing treats each element of an array individually, whereas static slicing considers each definition or use of any array element as a definition or use of the entire

array [103]. Similary, dynamic slicing determines which objects are pointed to by pointer variables during a program execution.

To compute dynamic slices, Korel and Lasky [85] proposed an iterative algorithm based on data-flow equations. In the case of loops, the algorithm requires that if any occurrence of a statement within a loop in the execution trace is included in the slice, then all the other occurrences of that statement in the trace will be included in the slice. This ensures that the slice extracted is executable. Other algorithms proposed in the literature produce slices that are not executable, because they are not necessarily executable subsets of the original program [4]. In particular the algorithm by Agrawal and Horgan [4] uses dynamic-dependence-graphs to produce more refined slices. It considers only the occurrences of statements in the trajectory that effect the computation of the variables in the slicing criterion. Interprocedural slicing algorithms based on dependence graphs have also been proposed [107] as well as dynamic slicing algorithms in the presence of unconstrained pointers [2] and arbitrary conrol flow [78].

Besides debugging [72, 85, 3], dynamic slicing has been used for several applications, including software testing [106], software maintenance [77, 101], and program comprehension. A survey and comparison of dynamic slicing methods has been presented by Korel and Rilling [102].

2.1.3 Quasi Static Slicing

Quasi static slicing was the first attempt to define a hybrid slicing method ranging between static and dynamic slicing [116]. The need for quasi static slicing arises from applications where the value of some input variables are fixed while others may vary. A quasi static slice preserves the behaviour of the original program with respect to the variables of the

slicing criterion on a subset of the possible program inputs. This subset is specified by the possible combination of values that the unconstrained input variable might assume. Of course, in the case all variables are unconstrained, the quasi static slice coincides with a static slice, while when the value of all input variables are fixed, the slice is a dynamic slice. The notion of quasi static slicing is closely related to partial evaluation or mixed computation [70], a technique to specialise programs with respect to partial inputs, by specifing the values of some of the input variables. Constant propagation and simplification can be used to reduce expressions to constants. In this way the value of some program's predicates can be evaluated, thus allowing the deletion of branches which are not executed on the particular partial input.

Quasi static slicing has been applied for program comprehension in combination with other program transformations [59]. Quasi static slicing can be considered as an extension of work presented in [45], where partial evaluation is used to aid program comprehension. Combining partial evaluation with program slicing allows us to restrict the focus of the specialised program with respect to a subset of program variables and a program point.

2.1.4 Simultaneous Dynamic Slicing

An other form of slicing; which was introduced by Hall [54], computes slices with respect to a set of program executions. This slicing method is called *simultaneous dynamic program slicing* because it extends dynamic slicing and simultaneously applies it to a set of test cases rather than just one test case. A simultaneous program slice on a set of test cases is not simply given by the union of the dynamic slices on the component test cases. Indeed simply unioning of dynamic slices is unsound, in that the union does not maintain simultaneous correctness on all the inputs. Hall [54] proposed an iterative algorithm that starting from an initial set of statements incrementally builds the simultaneous dynamic slice, by computing at each iteration a larger dynamic slice.

Simultaneous dynamic slicing has been used to locate functionality in code. The set of test cases can be seen as a kind of specification of the functionality to be identified. This approach can also be seen as an extension of the approach by Wilde et al. [111, 105], where test cases are used to identify the set of source code statements implementing a functionality. Combining slicing with this approach results in a more precise identification of the functionality to be extracted.

2.1.5 Other Slicing Methods

This chapter has so far surveyed statement deletion based methods for programs written in procedural languages. A number of slicing resources are available on the web (a good entry point is Jens Krinke's webpage[1]), including large scale slicing research tools (see for example tools developed within the Wisconsin [2] and the Unravel [3] slicing projects. Most of the proposed applications of slicing are related to software testing and debugging and to software maintenance tasks, such as program comprehension and restructuring, for example the introduction of new distributed technologies calls for applications of slicing to program parallelisation or migration to distributed architectures. Mark Weiser mentioned this in his seminal paper [126] and in his foreword to [60] pointed out the need for producing a major research effort in this direction. At the present, few contributions have been proposed. An example is the method presented by Canfora et al. [19], where a control-dependence based slicing algorithm is used to decompose legacy programs into client-server components.

The wide spread use of objects-oriented and distributed technologies also calls for new

[1] http://www.infosun.fmi.uni-passau.de/st/staff/krinke/slicing/slicing.html
[2] http://www.cs.edu/wpis.html
[3] http://hissa.ncsl.nist.gov/ jimmy/unravel.html

slicing algorithms. Several contributions to the definition of slicing methods for object-oriented [84, 115], concurrent, and distributed [21, 81] software applications have already been published, but there is little on slicing web-based applications developed with heterogeneous programming languages and technologies apart from [44].

A final point of concerns is the meaning of the term *slicing*. The framework of statement deletion based slicing methods [17] can be extended with more powerful simplification rules. Harman and Danicic [57] introduced *amorphous program slicing*. Like a traditional slice preserves a projection of the semantics of the original program from which it is constructed. However it can be computed by applying a broader range of transformation rules, including statement deletion. This is particulary useful in program comprehension, as more powerful transformation may sensibily simplify complex programs [13]. Other applications of this approach include the extraction of reusable functions, and program parallelisation. Indeed, it is worth noting that the slices extracted by removing irrelevant statements might cointain code fragments used for computing intermediate results and common to the different slices extracted from the original program [94, 51]. Transformation rules applied to these slices can make the extracted program components more self-contained and more understandable in future maintenance.

2.1.6 Applications

Debugging

Program debugging was the main motivation behind the introduction of program slicing by Weiser [124]. His motivation was a result of an observation he made that when debugging a program. He noted that programmers follows data and control dependencies to identify and locate the program statements responsible for the error [127]. During *debugging*, a programmer usually has a test case in mind which causes the program to fail. A

program slicer that is integrated into the debugger can be very useful in discovering the reason for the error by visualising control and data dependences and by highlighting that statements that are part of the slice. Variants of program slicing have been developed to further assist the programmer: program dicing [88] identifies statements that are likely to contain bugs by using information that some variables fail some tests while others pass all tests. Several slices are combined with each other in different ways: the intersection of two slices contains all statements that could lead to an error in both test cases; the intersection of a slice A with a complement slice B excludes from slice A all statements that do not lead to an error in the second test case. Another type of program slicing is program chopping [69]. It identifies statements that lie between two points n and m in the program and will be affected by a change at n. This can useful when a change at n causes an incorrect result at m. Debugging should be focused on the statements between n and m that transmit the change of n to m.

A bug in a program can be detected as the result of the execution of the program with respect to some specific input. As result of this, program debugging was the main motivation of Korel and Laski [79] to think of another variant of program slicing, dynamic program slicing [79], which produces slices that preserves the behaviour of the original program with respect to a particular input. Dynamic slicing [79] produces more accurate and smaller slices in size than than those produced by using traditional static slicing [126]. For this reason, dynamic program slicing is a more suited technique to assist programmers in locating a bug, exhibited on particular execution path of the program [3, **?**].

Program Differencing

Program differencing [38], is a way of analysing an old and a new version of a program in order to determine which part of the new version represent syntactic and semantic changes. This information is useful as in incremental testing where only the components that have a different behaviour need to be tested. The main issue of program differencing

consists of the partition of the components of the old and new version in a such a way that two components are in the same category only if they have equivalent behaviours. Program slicing can be used to identify *semantic* differences between two programs. This can be done in two stages:

1. We first find all the components of the two programs that have different behaviour. This can be done by comparing, using the dependence graphs, the backward slices of the old and new programs.

2. The second stage is the find a program that captures the revised semantics behaviour . This can be done by taking the backward slice with respect to a set of all of the affected program points determined in the first stage.

Program Integration

Program integration is a technique which consists of merging program variants [14, 10, 64]. The main motivation of the program integration technique is when different versions of a program have been made and a bug-fix, for example, is needed for all of them. Given a program `Base` and two different variants, `A` and `B`, obtained by modifying separate copies of the Base. Program integration will determine whether the modifications interfere or not. If an integrated program that incorporates both sets of changes as well as the portions of the `Base` which are preserved in both variants is not created [64], then *Program differencing* [38] is used to identify the changes in Variants `A` and `B`. The program integration algorithm discussed below compares slices in order to detect equivalent behaviours.

Horwitz et al. [64] used the static slicing algorithm for single-procedure programs as the basis for an algorithm that integrates changes in variants of a program. The algorithm consists of the following steps:

1. First, construct the program dependence graph (PDG), of the Base, A and B. Let G_{Base}, G_A and G_B represent the PDGs of the Base, A and B respectively.

2. The sets of affected points of G_A and G_B with respect to G_{Base} are determined. These consist of vertices in $G_A(G_B)$ which have a different slice in G_{Base}.

3. A merged program dependence graph, G_M is constructed from G_A, G_B and the sets of affected points determined in (2).

4. Using G_A, G_B, G_M and the sets of affected points computed in (2), the algorithm determines whether or not the behaviours of A and B are preserved in G_M. This is accomplished by comparing the slices with respect to the affected points of $G_A(G_B)$ in G_M and $G_A(G_B)$. If different slices are found, the changes interfere and the integration cannot be performed.

5. If the changes in A and B do not interfere, then algorithm tests if G_M is a feasible, i.e., if it corresponds to some program. If this is the case, a program M is constructed from G_M. Otherwise, the changes in A and B cannot be integrated.

Software Maintenance

The main challenges in software maintenance are to understand existing software, and make new changes without introducing new bugs. *Decomposition slicing* [51] is a useful tool in making a change to a piece of software without introducing unwanted side-effects. It captures all computations of a variable's value and is independent of a program location. The decomposition slice for a variable v is the union of slices taken at critical nodes with respect to v. Critical nodes are the nodes that output the value of v and the last node of the program. The decomposition slices are computed for all variables of the program. The decomposition slice for variable v partitions the program into three parts:

The independent part contains all the statements of the decomposition slice ,with respect to v, that are not part of any decomposition slice taken with respect to another

variable.

The dependent part contains all statements of the decomposition slice, with respect to v, that are part of another decomposition slice taken with respect to another variable.

The complement parts contains all statements that are not in the decomposition slice taken with respect to v. The statements of the complement may nevertheless be part of some other decomposition slice taken with respect to another variable. The complement must remain fixed after any change made to statements of the decomposition slice.

In the same way the variable v can be partitioned:

Changeable if all assignments to v are within the independent part.

Unchangeable if at least one assignment to v is in the dependent part. If the assignment has been modified, the new value will flow out of the decomposition.

Used if it is not used in the dependent or independent parts but in the complement. The maintainer may not declare new variables with the same name.

The modification can be made as follows:

- Statements in the independent part do not have any effect on the computation of the complement. Therefore, they can be deleted from a decomposition slice.
- Assignments to the changeable variables may be added anywhere in the decomposition slice.
- New control statements that surround any statements of the dependency part will cause the complement to change.

The maintainer who tries to change the code only has to consider the dependent and independent parts of the program. The complement part is guaranteed then to be unaffected by the modification and therefore there is no need to be retested [51]. The only parts to be retested after the modification are the dependent and the independent parts. There has been much work in using program slicing in software maintenance [18, 25, 50, 51].

Testing

After any modification, software has to be retested. A large number of test cases may be necessary, even after a small modification. Decomposition slicing can be used to reduce the program to a smaller one, which is easy to test, i.e. regression testing on the complement is not needed. Only the dependent and independent parts need to be to be retested. Extensive work has been done to simplify testing, using program slicing [12, 53, 56, 62, 63]. In [63], we have illustrated the use of program slicing in partition testing. Using program slicing to determine components affected transitively by a change in a program point p, Gupta et al. [53] introduced an algorithm for reducing the cost of regression testing. They use two variants of slices to achieve their goal. They first work out a backward slice starting from the program point p and record all the definitions of the variables used at p. They then work out a forward slice starting from the same point. Any variable which is defined at P and used in this slice is recorded. Finally Def-Use pairs from a definition of the first slice to a use in the second might be affected by the change at p and hence, they must be retested.

2.2 Symbolic Execution

2.2.1 Motivation Behind different Approaches and General Techniques

King in his Ph.D. thesis in 1969 was probably the first to introduce the concept of symbolic execution [75]. Since then, a number of others have developed similar methods, but for several different purposes, Depending on the emphasis of their work, some authors therefore talk about symbolic evaluation or symbolic testing instead of symbolic execution.

Symbolic execution can be used for verifying programs in a number of different ways, these methods are described below. An alternative use of symbolic execution is the generation of test cases, also described below. Both of these applications are often based on path analysis.

Path analysis

All of the early systems for symbolic execution, such as *EFFIGY* or *DISSECT*, see subsection 2.2.2, are based on the use of path analysis. When symbolically executing a program, one considers the different paths through the program separately. At any time during symbolic execution of the program, a path condition and a path value are associated with each path. The path value consists of the current (symbolic) values of the program variables, while the path condition consists of the condition on the input values under which this path is executed. Both path value and path condition are computed incrementally by symbolic execution of the program along the path. Whenever a statement changing the values of program variables is encountered, the path value is updated accordingly.

Whenever a branching statement is encountered, one branch is chosen and the appropriate branching condition is added to the path condition. The user may have to make an explicit choice which branch of a branching statement to take; alternatively, the symbolic execution system may automatically take all possible branches in turn. Obviously, the latter is in general not possible for loops, since they may give rise to an infinite number of paths. A particular path through a program and its associated choice of branches at branching statements is sometimes called a *test case*, for example in *EFFIGY* and in *DISSECT* if adding a branching condition to the path condition causes the path condition to become unsatisfiable, then that particular path will never be executed, whatever the input values, and the path can therefore be ignored in any further symbolic execution. This can be valuable information to the programmer since it might point to a mistake in the program. However, it does not necessarily mean that the appropriate branch is never taken, it may be taken when execution arrives at the same program point along a different path.

Program verification and validation

There are a number of different verification and validation methods based on symbolic execution. When using Hoare style inference rules for describing the semantics of a language, one can additionally provide assertions about input, output and loop invariants, and then generate verification conditions by symbolically executing the code between pairs of (adjacent) assertions. [55] describes this approach in detail. This can actually be very similar to what is done in path analysis. The main difference is that in order to handle infinite parts of the execution tree, in particular loops, one uses induction. In this context, induction comes in two different forms. The first, as introduced by [48], uses inductive assertions annotating a program. Such an inductive assertion states that whenever execution reaches the annotated point in the program, the assertion holds. This approach uses induction on the computation.

The second way of using induction was described in [16]. Burstall uses induction on the input data. In this approach. an assertion says that there exists a state during execution which is at the annotated point and satisfies the assertion.

A shortcoming of many current formal methods approaches to software development is the fact that failure to verify a given program often does not provide enough information about the cause of the failure; is it due to an actually incorrect program, or is it perhaps due to inappropriate assertions (in particular loop invariants), or has an existing correctness proof just not been found yet? To some extent. this problem can be overcome by symbolic execution, which can help to find the place in a program where it goes wrong, if this is indeed the case, or help to find appropriate assertions. This is why the symbolic execution system *SELECT* see subsection 2.2.2. for example, is described as a system for the debugging of programs.

Another technique to validate a program using symbolic execution is to annotate the program with error conditions in appropriate places and check for consistency with the path condition at these places. If the error condition is consistent with the path condition, then it is possible for this error to arise.

Test case generation

Other researchers consider symbolic execution as a method for test-case generation [68, 28]. The usual way of generating test cases using symbolic execution relies on a heavily simplified version of symbolic execution, based on path conditions only. This approach consists of generating the path conditions, but not the symbolic values associated with them, and then selecting a particular value for each path considered, i.e. finding a solution to the path condition. One usually tries to ensure at least branch coverage (every branch

at a conditional statement is covered at least once) when choosing these paths.

2.2.2 Other Symbolic Execution Systems

EFFIGY

Starting in 1973, King developed the system *EFFIGY* [75, 55, **?**] as a tool for program verification and validation. It supports symbolic execution of a simple PL/l-like language, and already includes most of the ideas used in later systems. In particular, it was based on the idea of path conditions, i.e. when symbolically executing a program, *EFFIGY* generates a path condition for every path traversed, and associates it with the symbolic or path value computed.

There are two ways of dealing with branching statements. In manual mode, the user has to decide at branching statements (*if-then-else or while loops*) which of the possible branches to take. In this case, the user may first save the state and come back to it later in order to explore a different branch. In automatic mode, on the other hand, all possible branches are explored. Loops are dealt with by allowing the user to specify a maximum number of computation steps. Only those paths are considered that reach an exit point within this number of steps. If, after the specified number of steps, no exit point has been reached, then that path is discarded and the system backtracks to the last branching statement with unexplored branches.

EFFIGY also allows the user to provide assertions at various points in the program. These are then used to generate verification conditions for the program.

GIST

GIST is a specification 'automatic programming' system being developed by Balzer and his colleagues at USC/ISI. One of the features provided by *GIST* is its symbolic execution facility [7, 33, 31] .

The overall idea of the *GIST* project is to build a programming environment based on transformations. First, a system is specified in the *GIST* specification language in terms of objects and relations between them. It is then implemented by gradually transforming it into a LISP program. Some of the transformations used for this are done fully automatically, others require the programmer to choose from a library, e.g. "do you want this set to be implemented as a list or a tree?". If the specification is changed at a later stage, part of it can be reimplemented fully automatically.

Since *GIST* specifications are expressed in terms of entities and relationships, they can neither actually nor symbolically be executed in the usual sense. The notion of paths and path conditions that can be used for programming languages cannot be applied in this case. *GIST* solves this problem by considering a specification as a set of axioms in a first order temporal logic. Symbolic execution then generates simple new theorems from these axioms, using a set of heuristics described in [32], these new theorems are then examined to decide whether they are 'interesting'. Only the interesting ones are displayed to the user.

GIST is supported by a paraphraser which translates *GIST* specifications into English in order to make them easier to understand [113]. This is then extended to explain the results of symbolically executing a *GIST* specification [114].

Work by Kemmerer, Rudnicki and Eckmann

In [73] , Kemmerer described a system for symbolically executing specifications, and compares this approach with the use of prototyping tools for specification validation. A simple version of this symbolic execution tool has been built, which can be used for symbolically executing specifications in INA J0[4]. The INA Jo specification language is a non-procedural language that models a system as a state machine, using the language of first-order predicate calculus. State transitions are described by post-conditions which are called *transforms*; these describe state transitions by specifying the values of state variables after the transition in terms of their values before the transition.

In [104], Rudnicki discusses the relationship between symbolic execution, as described by Kemmerer, and theorem proving for validating specifications. He emphasises the need to prove certain properties of a specification, in particular the preservation of criterions (invariants). Since he only considers the form of symbolic execution based on path analysis and not those forms used for verification and proving theorems about a program, he emphasises that symbolic execution (or *symbolic testing*, as he calls it) is not sufficient. In his opinion, proofs of properties of a specification should be done by hand rather than automatically (but supported by a proof checker), since failure to prove a statement using an automatic theorem prover does not in itself give many clues to why the proof failed - He does not consider the possibility of an interactive theorem prover that helps the user to find a proof. Next, Rudnicki suggests a symbolic execution strategy that seems to correspond roughly to a branch coverage strategy in testing:

> ”The minimal symbolic testing of a transform consists of testing each change of the state variables which can be caused by the transform at least once. The important thing is that each possible change of the state variable should be tested by a symbolic run starting in the initial state.” [104, page

[4]INA JO is a trademark of System Development Corporation, a Burroughs Company

192]

The initial state mentioned here is defined as part of the specification. With this strategy, one therefore has to build, for every change of state variables in a transform, a sequence of transforms starting in the initial state such that it reaches the relevant part of the transform.

In [74] , Kemmerer and Eckmann describe a different approach to symbolic execution which was implemented in the *UNISEX-system*. The original version of *UNISEX* was implemented by Solis as part of his master's thesis [109]. *UNISEX* is a system for symbolically executing PASCAL programs. It provides two modes, a *verify* mode and a *test* mode. Symbolic execution in the verify mode is based on the ideas described in [55] for verifying programs. To verify a PASCAL program, the user has to annotate it with a number of assertions. At a minimum, there has to be an entry, an exit assertion, and an assertion for each loop (the loop invariant). *UNISEX* then generates the relevant proof obligations by symbolically executing the code between two assertions. This can be driven either automatically, or manually. In the latter case, the user has to decide which branch to take at a branching statement. In the former case, *UNISEX* covers all branches by pushing the false branch on a stack and continuing along the true branch. When the end of the path is reached at an assert or exit statement or the end of the program or subroutine being verified, then another branch is popped from the stack and executed symbolically.

In its test mode, *UNISEX* uses path analysis to generate path conditions and path expressions. No assertions are needed, and even if they are provided, paths end at the end of the program rather than at the next assertion. Output from UNISEX is mainly intended to show the behaviour of the program and thus validate it, rather than formally verify the program against some assertions.
The expression language of UNISEX used for assertions consists of the expression language of PASCAL plus the additional keywords *forall*, *exists* and *implies*. Assertions are

provided as comments in the program to be executed symbolically, so that no editing is needed before the program can be compiled. The programming language supported is a sub-language of PASCAL as defined in [129] , excluding for example subroutines with side-effects.

DISSECT

DISSECT is another system of symbolic execution which uses path conditions[5] [66]. It tries to solve them by trying to find an actual value as *test case*. *DISSECT* was implemented in LISP and can be used to symbolically execute FORTRAN programs. The DISSECT symbolic evaluator takes as input a FORTRAN source program and a file with DISSECT commands that are to be applied to the program. There are three kinds of commands:

(1) Input commands, used to assign actual or symbolic values to variables;

(2) Path selection commands, used to determine which branch to take at a branching statement, or how many times to execute a loop;

(3) Output commands, used to print out the values of program variables.

Additionally, these commands can be combined in various ways, for example using conditionals. Every command is associated with a particular line in the FORTRAN program.

Every symbolic evaluation then effectively explores one path (called *test*) through the program. by determining its path condition and expressing the values of program variables at any point in the program in terms of the values assigned via input commands, it is possible to combine different paths (e.g. both branches of a branching statement), but

[5]This is the reason why the system is called DISSECT, it allows the user to 'dissect' a program and analyse the different paths separately.

these are effectively handled as different symbolic evaluations.

DISSECT runs in batch mode rather than interactively. According to [66] this is a deliberate choice, since the selection of these paths or tests has to be done carefully and therefore would be difficult to do interactively. In [67] , Howden analyses a number of methods for the validation of programs, including symbolic execution. There he applies these methods to six buggy short programs written in different languages (COBOL, PL/l and ALGOL). His conclusion is that symbolic execution helps to find about 5% of errors in addition to those found by combining various methods of testing, and is a *natural* way of discovering errors for about 10- -20% of all errors. The errors found by symbolic execution are mainly those where a wrong variable is referenced.

Dannenberg and Ernst's system

Dannenberg and Ernst describe another system for symbolic execution which is also based on path conditions [39]. It grew out of a project to design and implement a mechanical verification condition generator, although it is not clear from [39] whether this system has actually been implemented.

Dannenberg and Ernst use inference rules to describe the semantics of a small imperative programming language. In addition to the usual constructs, this language contains a *confirm* construct for expressing assertions and a *maintaining* argument in while-loops for expressing loop invariants. The *confirm* and *maintaining* constructs can be thought of as part of the specification of the program. The basic unit of the inference rules is a statement of the form $'S, PC \setminus A$ which expresses the correctness of the statement list A with respect to its specification, given a state S and initial path condition Pc.

Using these inference rules, Dannenberg and Ernst then propose to symbolically execute the code in between pairs of assertions to generate verification conditions. They use symbolic execution solely as a tool for verification condition generation, not for for directly generating infornation about the state and the path condition for the user.

An important feature of their work is that it shows a possibility of handling functions with side effects. For this purpose, they introduce attribute grammars where the state and path condition are represented as attributes of rules that describe symbolic execution. They also give rules for handling more complicated language constructs, such as multiple-exit loops and procedures with multiple exits. Unfortunately, they do not explicitly address the issue of the correctness and completeness of their rules.

ATTEST

The *ATTEST* system [26, 28] is a symbolic execution system mainly intended for generating test data for FORTRAN programs. It treats path conditions as a system of constraints. If the path condition only contains linear constraints, then *ATTEST* solves it, using a linear-programming algorithm. The result is then used both for identifying (in)feasible paths and for generating a set of test cases that satisfy branch coverage. [26] claims that this actually covers most practical cases, that is, most constraints in practice actually are linear.

ATTEST is not only meant for test data generation, but for program validation in a more general sense. This is why ATTEST actually builds up a symbolic representation of the program output associated with each path condition, which would not strictly be necessary for test data generation. ATTEST also helps to generate error conditions (e.g. for division by zero) and check them by adding them to the path condition and checking for consistency. However, since this consistency check is also based on the linear program-

ming algorithm, it can only handle linear path conditions and linear error conditions.

Loops are handled by trying to make their effect explicit by first expressing variable values recursively in terms of values on previous iteration (using so-called recurrence relations), and then eliminating this recursion, i.e. solving the recurrence relation. This is done by introducing a new variable denoting the number of executions of the loop. The result is then used to create a loop expression which replaces the loop itself. Unfortunately, in general this will not work, since many functions can only be expressed using recursion or loops that cannot be handled by ATTEST [28, page 147]. In these cases, the user has to explicitly specify the number of iterations of the loop. Furthermore, like most systems ATTEST can only handle a single path at a time, at branching statements the user has to make an explicit choice about which branch is to be taken.

SELECT

SELECT is a symbolic execution system developed by SRI [15]. The main purpose of S*ELECT* is the debugging of programs. It is intended to complement mechanical program verification and overcome some of the theoretical and practical problems associated with this approach. *SELECT* is built from the expression-simplifier part of the SRI Program Verifier by adding facilities for the symbolic execution of programs written in a subset of LISP.

Like most similar systems, *SELECT* is based on the notion of path conditions. At branching points, the system considers all feasible branches and the appropriate predicates are added to the path conditions. For loops. the user decides on the maximum number of iterations she wants the system to take. For each path, SELECT tries to maintain an example input. i.e, a solution to the path condition. This solution can then be used as actual test data for the program.

In addition to generating (simplified) symbolic values of program variables and actual input test data, SELECT also allows the user to annotate a program with assertions; according to [15], these can serve as

(1) Executable assertions. The assertion is a program in itself that is executed when the appropriate position in the program is reached.

(2) Constraints, which are simply added to the path condition. This enables the user to ensure that the test data generated satisfy some additional conditions.

(3) Checkpoints; when execution reaches such a checkpoint, the negation of the assertion is added to the path condition, and the system checks the result for consistency by trying to generate a solution to it, using the general solver for path conditions.

Harvard Symbolic Evaluator

The symbolic evaluator is a central part of the *HARVARD* Program Development System PDS. PDS supports programming in the EL1 language, an imperative programming language which supports constructs such as records, pointers, recursive procedures, and several ways of sharing of variables, in addition to the usual assignments, and loops etc.[6]

The basic idea behind the system is to derive semantic information about a program independently from its use in any tools such as those for exception detection, program verification and validation. The advantage of this approach is that semantic analysis is only performed once, instead of each tool performing its own analysis. Additionally, this guarantees that all tools assign the same semantics to the constructs of the programming

[6]See [99] for a description of the role that symbolic evaluation plays within PDS, and [71] for the technical details of the symbolic evaluator

language while, on the other hand, it is easier to adapt the tools to a different programming language, since only one analyser has to be rewritten.

This semantic analysis is based on symbolic evaluation. and the results of the analysis are then stored in a *program database*. Other tools can use this information to reason about the program without having to re-analyse the program, and can add further information which they deduce to the data base. Note that with this approach symbolic evaluation is used as a static analyser, not a dynamic interpreter.

[71] describes in some detail the simplification of expressions in the context of symbolic evaluation. The emphasis on symbolic evaluation providing input for further analysis by other tools, not by humans. This means that a *simpler* expression might be considerably more difficult to read for the human user. The simplifier generates suitable normal forms of expressions, based partly on CNF.

Loop analysis entails solving various recurrence relations. This involves identifying the number of iterations of a loop, and expressing the values of variables after any one iteration in terms of their values after the previous iteration. From this, one then tries to *solve* the recurrence relation, i.e. transform the recursive equation into an explicit one. If this is not possible, one *forces* a solution by creating a particular form of lambda expression. These recurrence relations also help to generate loop invariants automatically, at least in some easier cases.

REDUCE

REDUCE, as described in [5] , is a system for program reduction based on symbolic execution. This is a program transformation technique for removing superfluous parts of

a given program while leaving the original structure intact. For example, if a program contains a conditional with expression $x > 0$, and it subsequently becomes known that the program will only ever be used on input values such that $x > 0$, then the conditional may be replaced by its *then-part*, the *else-part* may be removed. *REDUCE* supports the reduction of such conditionals for a class of functional languages. Its main achievement, according to [5] , is that it allows symbolic constants which are assumed to denote subsets of data domains which allows one to express certain constraints on the input domain, These symbolic constants are predicates which are used to express input and path conditions. Note that there were other systems, such as *EFFIGY* that allow a user to add assertions on the input domain, with a similar effect.

REDUCE does support a certain amount of genericity with respect to the language handled, it can symbolically execute a whole class of functional languages. This is achieved by defining the *symbolic semantics* of a language using predicate transformers [43] on the symbolic constants.

2.3 WSL and FermaT

The *ConSUS* system is implemented using WSL, the Wide Spectrum Language introduced by Ward [118] for reverse engineering [119, 121, 122]. There were two main reasons why WSL was chosen as the basic language for this project:

1. Unlike conventional languages, WSL contains its own built in program transformation system *FermaT*. The *FermaT* `Simplify` transformation could be used as the basis for a light-weight validity checker.

2. Within WSL is another language `MetaWSL`, which facilitates parsing and manipulation of syntax trees. This would allow us to express the transformations used in

conditioning in a high-level and fairly natural way.

2.3.1 Theoretical Foundations of *FermaT*

The theoretical work on which *FermaT* is based originated in research on the development of a language in which correctness proofs for program transformations could be achieved as easily as possible for a wide range of constructs.

The WSL language was developed in parallel with a transformation theory and various proof methods. During this time the language has been extended from a simple and tractable kernel language [118] to a complete and powerful programming language. At the 'low-level' end of the language there are automatic translators from IBM assembler, TPF assembler, a proprietary 16 bit assembler, x86 assembler and PC code into WSL, and from a subset of WSL into C, COBOL and Jovial. At the 'high-level' end it is possible to write abstract specifications, similar to Z and VDM.

The use of first order logic in WSL means that statements can include existential and universal quantification over infinite sets, and similar (non-executable) operations.

The language includes constructs for loops with multiple exits, action systems, side-effects etc. and the transformation theory includes a large catalogue of proven transformations for manipulating these constructs. In [120] program transformations are used to derive a variety of efficient algorithms from abstract specifications. In [119] the same transformations are used in the reverse direction: using transformations to derive a concise abstract representation of the specification for several challenging programs.

WSL is defined in a series of stages or levels, with the lowest level being an extremely simple and tractable "kernel" language whose syntax is based on infinitary logic, and whose semantics is defined denotationally. In contrast to other work, WSL does not have a purely applicative kernel; instead, the concept of state is included, using a *specification statement* which also allows specifications expressed in first-order logic as part of the language, thus providing a genuine wide spectrum language.

Much work on WSL has been motivated by transformation-based reverse engineering, where it is essential to be able to represent in WSL the original legacy code, no matter how unpleasant. A state-based approach has meant that representation of low level constructs typical of such code is fairly direct. More recent work [121] has demonstrated the benefits for representing PLC (programmable logic controller) code and migrating x86 embedded systems to generic C code.

2.3.2 The *FermaT* `Simplify` Transformation

The source code for the *FermaT* `Simplify` transformation is very simple: it just calls the `@Simplify` function on the current item, then it invokes the `Simplify_Item` transformation on each component statement for which it is valid, then it invokes `Delete_Item` on every component statement for which it is invalid (other than assertions and comments). Finally it deletes `SKIP` statements within the current item:

```
MW_PROC @Simplify_Code(Data) ==
  @Paste_Over(@Simplify(@I, @Budget));
  FOREACH Statement
  DO
    IF @Cs?(@I)
      THEN IF @Trans?(TR_Simplify_Item)
```

```
          THEN @Trans(TR_Simplify_Item, "")
 FI
    FI;
   IF @ST(@I) <> T_Comment AND @ST(@I) <> T_Assert AND
      @Trans?(TR_Delete_Item)
   THEN @Trans(TR_Delete_Item, "")
   FI
 OD;
IF @Trans?(TR_Delete_All_Skips)
THEN @Trans(TR_Delete_All_Skips, "")
FI .;
```

Deleting a comment is always a valid transformation, but should not be carried out unless explicitly selected by the user.

All the real work of *FermaT* `Simplify` is carried out by the `@Simplify` function. This takes a syntactic item and a 'budget' (an integer value which indicates how much effort should be expended in trying to simplify the item) and returns a new item.

The requirements for this expression and condition simplifier were as follows.

1. To be efficient execution: especially on small expressions. This implies a short start up time.

2. To be easily extendible. It would be impossible to attempt to simplify *all* possible expressions which are capable of simplification. For example, it is known, now that Fermat's Last Theorem has been proved, that the integer formula $n > 2 \wedge x^n + y^n = z^n$ can be simplified to *false*, but it cannot be expected that an automated procedure

should be able to prove it from first principles. Since we must be content with a less-than-complete implementation, it is important to be able to add new simplification rules as and when necessary.

3. To be easy to prove correct. Clearly a faulty simplifier will generate faulty transformations and incorrect code. If the simplifier is to be easily extended, then it is important that the extended simplifier can be proved correct equally easily.

In order to meet requirement (2) the heart of the simplifier is table-driven, consisting of a set of pattern match and replacement operations. For example, the condition $x + y \leq z + y$ can be simplified to $x \leq z$ whatever expressions are represented by x, y and z. This pattern match and replacement can be coded as a simple `ifmatch` and `fill` in `MetaWSL`. To reduce the number of patterns required, the simplifier first normalises the expression as follows.

1. If the current item is neither an expression nor a condition, then `@Simplify` is invoked recursively on all its components;

2. Otherwise, the first step is to push down negation operations by applying De Morgan's Laws. For example $\neg(A \vee B)$ is transformed to $\neg A \wedge \neg B$ and $\neg(a = b)$ becomes $a \neq b$;

3. Then the function flattens any associative operators by removing nested parentheses, for example $((a + b) + c)$ becomes $(a + b + c)$. Subtraction and division operators are replaced by the equivalent negation and invert constructs, for example $a - b$ becomes $a + (-b)$.

4. The next step is to evaluate any components which consist entirely of constants;

5. Then sort the components of commutative operations and merge repeated components using the appropriate power operator. For example, $a+b+a$ becomes $2*a+b$ while $a * b * a$ becomes $a^2 * b$;

6. Multiplication is expanded over addition, for example $a*(b+c)$ becomes $a*b+a*c$ and AND is expanded over OR;

7. The steps from step 3 are repeated until the result converges

The next step is to check each pattern in the list. If any of the patterns matched, then repeat from step 2 with a reduced budget until the result converges or the budget is exhausted. Finally expressions are factorised where possible and then some final cosmetic cleanup rules are applied: for example, people usually write $2 * x$ (putting the number first in a multiplication operation), but $x + 2$ (putting the number *last* in an addition).

2.4 The Co-operating Validity Checker (CVC)

CVC stands for the Cooperative Validity Checker. This tool has been in development at Stanford for several years, and is used both in research, and as a formal hardware verification tool, as seen in such papers as [112]. CVC allows users to check formulae based on a subset of first order logic and boasts of its efficient and automatic decision procedures.

The logic for these decision procedures includes booleans, uninterpreted functions and linear arithmetic. There are also uninterpreted functions such as array operations, bit vectors and parts of linear arithmetic.

CVC provides a variety of commands to the user, from a traditional command-line user interface. At its simplest level, proof is carried out through use of the cvc command, which is given an expression in CVC logic, and will return an answer based on the validity of that expression, and possibly a counter example, if the expression is found to be invalid. CVC also provides numerous other commands which aid in proof. These can relate to

various command flags, which affect the way CVC works, or manipulates CVC's context stores proven results.

CVC reads input in a presentation language. The input is a sequence of commands. Each command ends in a semi-colon.

CVC's presentation language is typed. In a typical example, constant symbols, function symbols, and predicate symbols are declared, with their types. Then some formulas are asserted. Finally, some queries are posed, to determine whether or not the asserted formulas logically imply some other formulas.

2.4.1 Commands

This section describes the commands of CVC's presentation language.

Type Declarations

New basic types can be declared in CVC using the following syntax:

```
t_1, ..., t_n :  TYPE;
```

For example, the following two CVC commands declare a, b, and c to be new basic types.

```
a :  TYPE; b, c :  TYPE;
```

Type Definitions

New types may be defined in terms of old ones using the following syntax:

```
t :  TYPE = t';
```

Here, it must be the case that given the previous definitions and declarations, t' is a well-formed type. t must be a new identifier. t becomes a name (or an abbreviation) for the type t'.

For example, the following defines rPair to be an abbreviation for the tuple type [REAL, REAL]:

```
rPair :  TYPE = [ REAL, REAL];
```

ASSERT

A formula can be added to the current logical context using the following syntax:

```
ASSERT F;
```

F must be a well-formed expression (given previous definitions and declarations) of type BOOLEAN.

For example, the following is a valid assertion (given suitable declarations or definitions of x, y, and z).

```
ASSERT x + 2 * y = z;
```

QUERY

CVC can be queried to determine whether or not a formula is logically implied by the formulas (if there are any) in the current logical context using the following syntax:

```
QUERY F;
```

F must be a well-formed expression (given previous definitions and declarations) of type BOOLEAN. CVC will output ”Valid.” if F is valid in the current logical context and ”Invalid.” otherwise (ASSERT adds formulas to the current logical context). If CVC reports that the formula is invalid, it will remain in the logical context where the invalidity was detected. In general, this context may contain formulas that were not in the logical context when the query command was issued. This is because CVC's propositional *SAT solver* dynamically adds formulas to the context.

It can be useful to assert some formulas, and then issue several QUERY commands. CVC may be able to process something like

```
ASSERT F_1;
QUERY F_2;
QUERY F_3;
```

faster than

```
QUERY F_1 => F_2;
QUERY F_1 => F_3;
```

because work performed to handle the assertion of F_1 will not be duplicated in the former

case, but may be duplicated in the latter.

TRANSFORM

Any expression can be simplified with respect to the current logical context using the following syntax:

```
TRANSFORM E;
```

Given previous definitions and declarations, E should be a well-formed expression (a type, term, or formula). The simplification does not necessarily reduce expressions to a canonical form.

DUMP_PROOF and DUMP_SIG

If CVC reports that a queried (with QUERY) formula is valid, this command will dump a proof of that formula. For *DUMP_PROOF* to be available, CVC must not be in SAT mode, and CVC must have been invoked with the command-line option *+proofs*.

DUMP_PROOF may be called like this:

```
DUMP_PROOF;
```

or like this:

```
DUMP_PROOF "F";
```

where F is the name of a file to which the proof should be written. In the former case, the proof will be written to *stdout*.

The proof produced will generally depend on the assertions (added with ASSERT) in the current logical context, and on the declarations of types and constants. Those declarations are dumped separately using the *DUMP_SIG* command, which may also be called with the name of a file to use or with no arguments.

DUMP_ASSUMPTIONS

DUMP_ASSUMPTIONS is available under the same conditions as DUMP_PROOF, and may be called in the same way. It dumps all the assumptions that were relevant to the validity of a queried (with QUERY) formula which CVC reports to be valid.

2.4.2 Types

CVC's presentation language is typed. The type system is fairly simple. There is no polymorphism or subtyping. In addition to built-in basic types, there are tuple types, record types, array types, scalar types (aka, enumerations) inductive datatypes, and function types.

Basic Types

The basic types in CVC are REAL and BOOLEAN. REAL is the type of numbers, and BOOLEAN is the type of formulas.

Technically, since CVC's language is first-order, types should not contain BOOLEAN, except as the range of a function type (which is then the type of a predicate). In CVC, however, BOOLEAN may appear arbitrarily in types. This is achieved by having two

versions of the BOOLEAN type, one for formulas and one for boolean terms. This distinction is made only internally; the input language just has a single type BOOLEAN.

Tuple Types

The types for tuples are of the form

```
[ t_0, ..., t_(n-1) ]
```

where n is at least 2, and t_0, ..., t_(n-1) are types. For example, the following declares a, b, and c to be various tuples:

```
a :  [ REAL, REAL ];
b :  [ REAL, BOOLEAN, [ REAL, REAL, REAL] ];
c :  [ BOOLEAN, BOOLEAN, [ REAL, BOOLEAN] ];
```

Function Types

The types of functions are of the form

```
[ Domain -> Range ];
```

where Domain and Range are types. For example, this declares a function F from tuples to tuples:

```
F : [ [ REAL, REAL ] -> [ BOOLEAN, BOOLEAN ] ];
```

Record Types

The types for records are of the form

```
[# f\_1 : t\_1, ..., f\_n : t\_n #]
```

where n is at least 1; f_1, ..., f_n are identifiers; and t_1, ..., t_n are types. f_1, ..., f_n are called the fieldnames of the record type. The fieldnames f_1, ..., f_n may be the same as the names of fields of other records or other declared or defined constants and types.

Here is an example

```
f :  REAL;


r : [# f : BOOLEAN, f2 : REAL #];
```

Inductive Datatypes and Scalar Types

An example of an inductive datatype (also known as an abstract datatype, or ADT) is

```
DATATYPE cons (car :  REAL, cdr :  rList), null END
```

This is the type of finite lists of REALs.

The general form for inductive datatypes is

```
DATATYPE constructor_def_1, ...  , contructor_def_n END
```

where each constructor_def_i is of the form

```
c (s_1 :  t_1, ..., s_m :  t_m)
```

Here, c is the name of the constructor. s_1, ..., s_m are the names of the selectors (aka, destructors) for the constructor c. t_1, ..., t_m are the domain types for the constructor c. They are also the range types for s_1, ..., s_m, respectively. If m is 0, then the list of selectors is omitted from the constructor_def.

When a datatype is declared, CVC automatically declares each constructor c as the appropriate function; if c has no selectors, it is declared as a constant. The selectors for c are also declared as the appropriate functions. Applying the i'th selector of c to c(a_1, ..., a_n) returns a_i. Selectors are treated as partial functions in CVC.

CVC automatically declares a tester "c?", which is a predicate on the datatype. It returns true for a given element e of that datatype iff e was constructed using constructor c.

Scalar types are a special case of inductive datatypes. They are of the form

```
c_1, ..., c_n
```

where n is at least 1, and c_1, ..., c_n are new identifiers. As noted already, the use of inductive datatypes and scalar types is restricted in a way that the user of other types is not.

Array Types

The types for arrays are of the form

```
ARRAY I OF R
```

where I and R are types, and I is not allowed to be an array type.

This example declares a multi-dimensional array :

```
a :  ARRAY REAL OF ARRAY REAL OF ARRAY REAL OF REAL;
```

2.4.3 Terms and Formulas

The terms of CVC's presentation language are described below. Formulas are considered as terms of type BOOLEAN. Predicates are considered as functions returning BOOLEAN.

Propositional combinations

The operators AND, OR, and NOT are the usual propositional connectives. XOR is exclusive or, $\Rightarrow$ is implication, and $\Leftrightarrow$ is bi-implication. These connectives each take two BOOLEANs as arguments and return a BOOLEAN. Their applications are written in infix notation. For example, CVC will report that the queried formula below is valid:

```
p, q :  BOOLEAN;
QUERY (p OR NOT p) AND (q => (q <=> NOT (q XOR q)));
```

Equalities and Disequalities

Equations and disequations between terms are written in infix notation using the "=" and "/=" symbols, respectively. Equations and disequations are of type BOOLEAN. For example:

```
x, y :  REAL;
p, q :  BOOLEAN;
```

```
PRINT x = y AND p /= q;
```

Arithmetic Inequalities

Arithmetic inequalities between REALs are written in infix notation using $<$, $\leq$, $>$, and $\geq$, with the usual meanings. Inequalities are of type BOOLEAN. For example:

```
x, y :  REAL; TRANSFORM x < x OR x >= y;
```

Tuples

Tuples are of the form

```
( t_0, ..., t_(n-1) )
```

where n is at least 2 and t_0, ..., t_(n-1) are terms, with types A_0, ..., A_(n-1), say. The type of the tuple is then [A_0, ..., A_(n-1)]. The i'th component of a tuple T which has at least (i+1) components is selected using this syntax:

```
T.i
```

If the type of T is [A_0, ..., A_(n-1)], then the type of T.i is A_i. The numbering of the components starts at 0, as shown in this example, for which CVC will report valid:

```
T : [ REAL, REAL ];
ASSERT T.0 = (34, 35).1;
QUERY T.0 = 35;
```

Function Applications

The application of a function F to argument A is written

```
F(A)
```

In the common case where A is a tuple like (x,y), it is written simply as

```
F(x,y)
```

An application of a constructor c of an inductive datatypes to arguments x and y is also written as c(x,y).

Here is an example:

```
x, y :  REAL;
F : [ [ REAL, REAL ] -> REAL ];
G : [ REAL -> REAL ];
H : [ [ REAL, REAL ] -> BOOLEAN ];
PRINT F(x,y) = G(x) OR H(x,y);
```

Records

Record literals are of the form

```
(# f_1 := v_1, ..., f_n := v_n #)
```

where n is at least 1; f_1, ..., f_n are identifiers; and v_1, ..., v_n are values, with types A_1, ..., A_n. The type of the record literal is then

```
[# f_1 : A_1, ..., f_n : A_n #] (see record types).
```

As for record types, the fieldnames f_1, ..., f_n may be the same as the names of fields of other records or other declared or defined constants and types.

The component corresponding to a field f of a record R which has such a field is selected using this syntax:

```
R.f
```

CVC reports valid for this example:

```
a, b : REAL;
R : [# x : REAL, y : REAL #];
ASSERT R.x = a + b;
QUERY (# a := R.x #).a = b + a;
```

Arrays

To read from an array A at index i, the syntax is

```
A[i]
```

Arithmetic Operations

The symbols ”+”, ”-”, ”*”, ”/” are used for addition, subtraction and unary minus, multiplication, and division, respectively. They each take REALs and return a REAL. Except for unary minus, they are all binary operations and are written using infix notation.

REALs 0, 1, 2, ... are also present. CVC is complete only when one child of every multiplication and division is a constant arithmetic expression (like 3 or 3/4).

2.4.4 Semantic notes

Arithmetic Fragment

The decision procedure for arithmetic is complete only for a fragment of arithmetic over the reals. The operations allowed in this fragment are addition, subtraction, arithmetic comparison, multiplication by a numeral, and division by a numeral. So multiplication or division of two variables immediately is outside the decided fragment.

Partiality

Some functions in CVC are partial. For example, division is undefined if its second argument is 0. Selectors of inductive datatypes are also partial; car(null) is considered to be undefined.

Partiality is handled in CVC using the two-valued approach advocated by Farmer [117]. In this approach, terms may be undefined, but formulas are always either true or false. A predicate applied to an undefined term is considered to be simply false.

Here is an example demonstrating this approach:

```
x, y :  REAL;
ASSERT x = 1/y;
QUERY y /= 0;
```

CVC will report valid for this query, for the following reason. Under the two-valued

approach, the assertion of x = 1/y will be false if y is 0. This is so because if y is 0, 1/y will be undefined, and the predicate "=", like all predicates, returns false if applied to an undefined term. So for the assertion to be true, y must not equal 0. Hence, the queried formula is valid.

2.4.5 Checking Proofs with Flea

CVC comes with the flea proof-checker. When CVC reports that a formula is valid, it can optionally produce a proof of that formula, which can be checked by flea. Since flea is a much simpler tool than CVC, double-checking CVC's results with flea can greatly decrease the probability that CVC reported the formula valid erroneously.

Once a proof and a signature have been written to a file from CVC, the proof can be checked with respect to the declarations in the signature, using the script bin/run_flea.sh.

For example, suppose CVC is run on the following input:

```
x, y, z :  REAL;
ASSERT x = y;
ASSERT y = z;
QUERY x = z;
DUMP_SIG "sig.flea";
DUMP_PROOF "pf.flea";
```

The following command will then run flea to check the proof:

```
run_flea.sh sig.flea pf.flea
```

2.5 ConSIT

ConSIT is the the first fully automated implementation of conditioned slicing. It operates on a subset of C, for which a tokeniser and symbolic executor were written in Prolog.

The top level algorithm is quite simple. It is depicted in Figure 2.1. Phase 1 propagates state information from the condition in the slicing criterion, to all points in the program, using symbolic execution. Phase 2 produces a conditioned program by eliminating statements which are never executed when the initial state satisfies the condition mentioned in the slicing criterion. These are precisely those for which the state information defines an inconsistent set of state. The test of consistency of each set of states is computed using the `Isabelle` theorem prover, as described in more detail in Section 2.5.2. Phase 3 removes statements from the conditioned program which do not affect the static part of the conditioned slicing criterion. Phase 3 is implemented using the *Espresso* static slicing system [37].

Phase 1:	Symbolically Execute
Phase 2:	Produce Conditioned Program
Phase 3:	Perform Static Slicing

Figure 2.1: The three phases in slice construction

The system is built from various components written in different languages. The symbolic executor and conditioner were written in Prolog. These were developed to work on an imperative programming language called `Haste` that includes loops, input statements and conditionals. The conditions of Haste are defined to be a good match for legal `Isabelle` propositions, and it is easy to parse using a Definite Clause Grammar within Prolog.

The `Isabelle` theorem prover [97, 98, 96] is used to check the reachability of a statement. This is written in Standard ML. A wrapper script written in `Expect` acts as an `Isabelle` server process. It runs the theorem prover on a pseudo terminal and listens

for connections on a socket. When the conditioner requires the services of `Isabelle`, it spawns an `Expect` client process which connects to the `Isabelle` server, via its socket, and requests that the symbolic state be analysed (using `Isabelle`'s auto-tacticals). The server then interacts with `Isabelle` over the pseudo terminal, returning the result of the query back to the client process. The exit code of the client process is then used to indicate the result of the query.

A similar client-server approach is adopted with the slicer. However, in this case the slicer and its server process are written in `Java`, based upon the *Espresso* slicing system [37]. *Espresso* uses a parallel slicing algorithm [38] which takes advantage of inherent CFG parallelism, where each node of the CFG of the subject program is compiled into a separate `Java` thread.

A pre- and post- processor, written in `JavaCup` and `JLex`, are used to translate between C and the internal language Haste.

2.5.1 The Symbolic Execution Phase

The symbolic states consist of disjunctions of conditional states, each of which in turn is a pair consisting of (in)equalities that arise from conditional expressions and equality statements that arise from assignment and input statements.

In practice, symbolic states are represented as sets of *pairs*. The first element of each pair is a set of bindings between variables and their symbolic values that arise from assignments and input statements. Canfora et al [17] call this the *symbolic state*. The second element of each pair is a set of (linear) inequalities. Canfora et al [17] call this the *path condition*. The intended interpretation is that each pair corresponds to a conditional symbolic state; the variables will have the symbolic values given in the first element of the pair when the (in)equalities in the second element are true.

The symbolic executor is derived directly from the semantics of the language. The semantics $\sigma(S)$ of a sequence of statements s is a set of states $\{s_1, s_2, \ldots, s_n\}$. Each symbolic state s_i ($1 \leq i \leq n$) can be considered as the pair[7]:

$$\langle a_i, c_i \rangle$$

Each a_i is an assignment function from variables to expressions, and c_i is a path condition. $\sigma(S)$ denotes the effect of symbolically executing S, replacing each variable reference with its current symbolic value (or a unique skolem constant, indicating an unknown symbolic value).

Each element of a_i is an assignment of the form $\langle v, e \rangle$, intended to mean that variable v is assigned the value of the expression e. The conditions c_i indicate the inequalities that must hold between the values of variables and other expressions for the program to be in a state of the form a_i. $\sigma(S)$ is called the *symbolic (conditional) states* of S.

2.5.2 The Theorem Proving Phase

The symbolic execution phase takes a sequence of program statements and annotates it with symbolic state descriptions. The implementation seeks to simplify these symbolic states by eliminating those conditional states that are inconsistent from each symbolic state description to determine the paths through the structure of the program code that can never be taken. The theorem prover is thus used to determine whether the outcome of a predicate *must* be `true` or whether it must be `false` or whether it is not possible to tell. This process is inherently conservative, because there will be predicates which must be `true` and those which must be `false` but for which this information cannot be deduced by the theorem prover. However, this conservatism is safe: if a statement is removed because of the outcome of the theorem proving stage, then that statement is

[7] The notation, $\langle x, y \rangle$ is used for pairs, to aid the eye in distinguishing pair constructions from parenthetic sub-terms.

guaranteed to be unnecessary in all states which satisfy the initial condition mentioned in the conditioned slicing criterion.

Using this symbolic execution semantics, each statement in the program is associated with the set of all conditional contexts, $\{\langle a_1, c_1\rangle, \ldots \langle a_n, c_n\rangle\}$, in which that statement could possibly be executed.

This set of contexts is transformed into a proposition using a function $\mathcal{P}$, where

$$\mathcal{P}\{\langle a_1, c_1\rangle, \ldots \langle a_n, c_n\rangle\}$$
$$=$$
$$(\mathcal{P}'(a_1) \wedge \bigwedge c_1) \vee \ldots \vee (\mathcal{P}'(a_n) \wedge \bigwedge c_n)$$

and

$$\mathcal{P}'\{\langle v_1, e_1\rangle, \ldots, \langle v_n, e_n\rangle\} = (v_1 = e_1) \wedge \ldots \wedge (v_n = e_n)$$

If the proposition is inconsistent (`false`), then it is inferred that the statement can never be executed: there is no path to the statement as each of the possible pairs of assignments and path conditions are inconsistent. The program is then equivalent to one in which the statement is replaced by the empty statement. In this way the conditioned program is constructed by considering which statements have inconsistent path conditions.

2.5.3 Complexity

Empirical tests indicate that the most time consuming aspect of conditioning is the validity checking of the symbolic semantics. This in turn depends upon the number of terms in the proposition to be checked.

As can be seen from the symbolic semantics, the size of the expressions produced increase for every statement. The number of paths is determined by the number of conditional statements and the number of while loops. For c conditional statements, and l loops, the number of paths is $\mathcal{O}(2^{c+l})$.

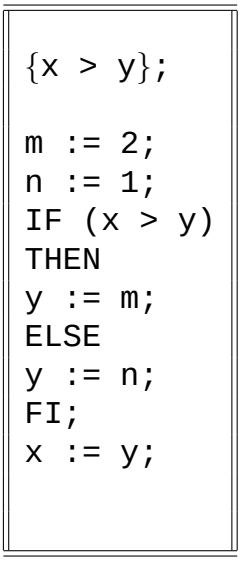
The number of atomic propositions within each path is the sum of number of assignment statements a on that path and the number of atomic propositions p within the boolean component of each conditional statement and loop $\mathcal{O}(a + p(c + l))$. So the size of the symbolic expression at the final statement of a program is $\mathcal{O}(2^{c+l}(a + p(c + l)))$, or $\mathcal{O}(2^n(a + pn)) = \mathcal{O}(2^n)$, where $n = c + l$.

In conditioning, the symbolic semantics is evaluated at each statement, not just at the end of the program. Considering just loops and conditionals, this gives $\mathcal{O}(2^0 + 2^1 + 2^2 + \cdots + 2^n) = \mathcal{O}(2^n)$.

2.5.4 Discussion

There are many ways in which the system described here can be enhanced, both in regard to performance and functionality.

Given the $\mathcal{O}$ complexity of the conditioning process, it is important to seek to reduce the complexity of the analysis were possible. One way in which this can be achieved is to "fold" the reasoning and symbolic execution processes together: as described *ConSIT* generates all possible paths to each statement, and has to check the accessibility of each one. *ConSUS* makes use of the monotonicity of the propositions that it has to analyse: if a path becomes infeasible, then it will remain infeasible for all subsequent statements. *ConSUS* "prunes" these paths once there inaccessibility has been determined. In some circumstances, this has a significant effect on the size of the propositions handed to the theorem prover.



```
{x > y};

m := 2;
n := 1;
IF (x > y)
THEN
y := m;
ELSE
y := n;
FI;
x := y;
```

Figure 2.2: A program fragment to be conditioned sliced

Another optimisation concerns the ordering of the conditioning and slicing processes in the larger system. Both slicing and conditioning effectively remove statements from a program. Slicing is not as complex as conditioning. In empirical tests, if we slice a program before we condition there is a dramatic improvement in overall performance. However if we slice a program first and then condition, we may have to slice it again. To demonstrate this, Consider the program in Fig 2.2. Slicing this program with respect to `x` produces the same program. We can simplify the program further by conditioning with respect to the condition `x>y`. This removes the *else* part as it is an infeasable path yielding the conditioned slice in Figure 2.3. However, this is clearly not the smallest conditioned slice of the program in Figure 2.2 as the statement `n:=1;` does not contribute to the final computation of `x`. To remove this statement, we need to slice the program in Figure 2.3 *again* with respect to `x`.

As described in section 2.5.2, *ConSIT*'s conditioner effectively just deletes inaccessible statements. Other simplifications are possible. For example, in the conditional expression

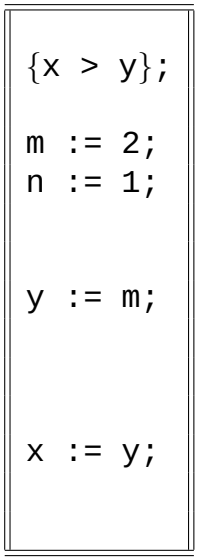


Figure 2.3: A conditioned slice of the program in Fig 2.2

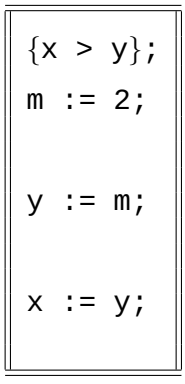


Figure 2.4: A *smaller* conditioned slice of the program in Fig 2.2

`if` c s_1 `else` s_2

if the s_1 (s_2, respectively) is inaccessible because c is always false (true, respectively), then the entire statement can be replaced by s_2 (s_1 respectively).

In a similar vein, with the loop

`while` c s

if we can show that c is always true after the first execution of s, then the entire loop can be replaced by s.

Chapter 3

Program Conditioned Slicing

3.1 Program Conditioning

Program *conditioning* is the act of simplifying a program assuming that the states of the program at certain chosen points in its execution satisfy certain properties. These properties of interest can be expressed by adding assert statements to the program being conditioned. Consider, for example the program in Figure 3.1. Here, program simplification is being attempted under the assumption that the program is executed in an initial state where `x>y`. In such states, the *true* path of the `IF-THEN-ELSE` statement will always be taken and thus the program can be simplified to `{x>y};a:=1`. Notice that the assert statement is also included in the resulting conditioned program. This is because an assert statement is a valid WSL statement that aborts if its condition is *false*. Observe that the conditioned program's behaviour is identical to that of the original program with the assert statement.

A software engineer may require a program to be conditioned with respect to intermediate states as well as with respect to initial states. The example in Figure 3.2 expresses the fact that the program is to be simplified assuming that all its inputs are positive. The conditioner should be able to replace the final `IF-THEN-ELSE` statement by `PRINT("POSITIVE")`.

```
{x>y} ;
IF x>y
THEN a:=1
ELSE a:=2
FI
```

Key

Conditioned program: boxed lines of code
Condition: x>y

Figure 3.1: Conditioning a simple program

```
i:=1 ;
total:=0 ;
WHILE i<n
DO
    INPUT(x) ;
    {x>0} ;
    total:=total+x ;
    i:=i+1
OD ;
IF total<0
THEN PRINT("NEGATIVE")
ELSE PRINT("POSITIVE")
FI;
```

Key

Conditioned program: boxed lines of code
Condition: x>0

Figure 3.2: Conditioning with intermediate asserts

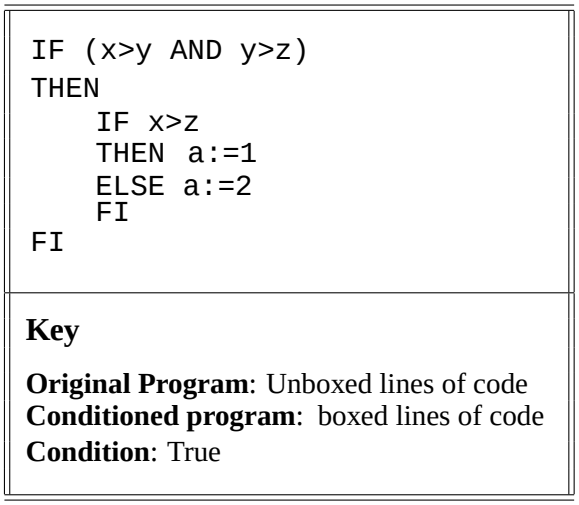


Figure 3.3: Conditioning without assert

A *program conditioner* is a program which tries to remove code that is unreachable given the assertions. Therefore, a conditioner will try to remove unreachable code even if the program contains no assert statements (see Figure 3.3 for an example of this).

Conditioners are required to reason about the validity of paths under certain conditions. In order to perform such reasoning, it may seem sensible to utilise existing automated theorem provers rather than to develop new ones. Consider the program in Figure 3.4, here, conditioning of a simple `IF` statement assuming that the initial state has the property that `x>y AND y>z` is being attempted. This is achieved by adding the corresponding assert statement at the beginning of the program. The simplification achieved depends upon the conditioner's ability to infer that $x > y \; AND \; y > z \implies x > z$. If the conditioner knows that the operator, $>$, is transitive, then it will be able to infer that the second of these conditions is a contradiction and therefore that the `ELSE` branch of the `IF` is infeasible. Only the Assert statement, {`x>y AND y>z`}, and assignment `a:=1` are required; the rest of the code can be removed.

The simplifying power of the conditioner depends on two things:

1. The precision of the symbolic executor which handles propagation of state and path information.

2. The power of the underlying theorem prover which determines the truth of propositions about states and paths.

By using an approximation to a program's semantics using a form of symbolic execution, and by being willing to accept approximate results from the theorem proving itself, conditioning allows us to adopt reasoning that does not require the full force of inductive proofs. The theorem proving used in program conditioning is lightweight when compared to the theorem proving required for a complete formal analysis of a program.

The problem can be further constrained to cases where the theorem proving can be implemented by completed decision procedures. There are limitations to the kinds of expressions for which complete decision procedures exist, one typical limitation is a restriction to reasoning with sets of so-called 'linear' (in)equalities. In our implementation of *ConSUS* we have experimented by incorporating two different reasoning systems into the same basic '*conditioning engine*'. These are:

- *FermaT* `Simplify` (WSL's own lightweight simplifier)
- The Co-operating Validity Checker (CVC) [112]

Although CVC is a very powerful (it is complete for linear arithmetic), we are using it in a light-weight manner. It turns out that the program in Figure 3.4 is an example of one that is simplified using *CVC* but not using *FermaT* `Simplify`.

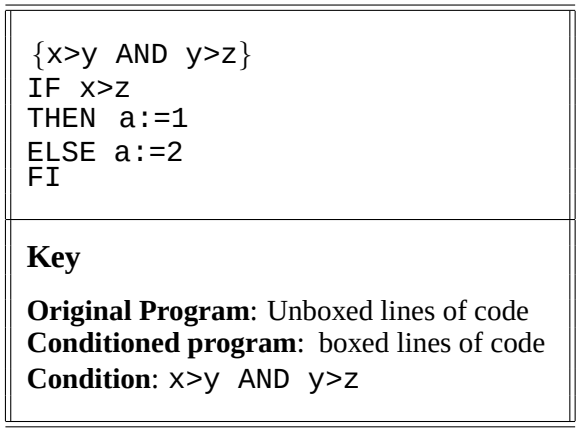


Figure 3.4: Conditioning a simple program using *CVC*

3.2 Conditioned Slicing

Conditioned slicing is a general framework for statement deletion based slicing [17]. A conditioned slice consists of a subset of program statements which preserves the behavior of the original program with respect to a slicing criterion for any set of program executions. The set of initial states of the program that characterises these executions is specified in terms of a first order logic formula on the input.

Conditioned slicing allows a better decomposition of the program giving human readers the possibility to analyse code fragments with respect to different perspectives. Canfora *et al.* [17] have demonstrated that conditioned slicing subsumes any other form of statement deletion based slicing method, i.e., the conditioned slicing criterion can be specified to obtain any form of slice.

A conditioned slice can be computed by first simplifying the program with respect to the condition on the input (i.e., discarding infeasible paths with respect to the input condition) and then computing a slice on the reduced program. A symbolic executor [75, 34] can be

used to compute the reduced program, also called a *conditioned program* in [18].

As an example of the way in which conditioning identifies sub-programs, consider the Taxation program in Figure 3.5. The figure contains a program fragment[1] which encodes the UK tax regulations in the tax year April 1998 to April 1999. Each person has a 'personal allowance' which is the portion of their income that is untaxed. The size of this personal allowance depends upon the status of the person, which is encoded in the boolean variables `blind`, `married` and `widowed`, and the integer variable `age`. For example, given the condition

```
age>=65 AND age<75 AND income=36000 AND blind=0
                AND married=1
```

conditioning the program identifies the statements which appear boxed in the figure. This is useful because it allows the software engineer to isolate a sub-computation concerned with the initial condition of interest. The extracted sub-program can be compiled and executed as a separate code unit. It will be guaranteed to mimic the behaviour of the original if the initial condition is met. Although the indentification of the infeasible path of condititioned programs is in general an undecidable problem, in many cases implications between conditions can be automatically evaluated by a theorem prover e.g [89]. In [17] conditioned slices are interactively computed: the software engineer is required to make decisions the theorem prover cannot make.

Different variants of conditioned slicing have been presented in the literature [80, 46]. Ning *et al.* [80] proposed a tool, called COBOL/SRE, to extract different types of slices from legacy systems, in particular conditioned-based slices. The user specifies a logical expression and a slicing range and the tool automatically isolates the statements that can

[1] This is a WSL version of the C program previously used in [36].

```
IF (age>=75) THEN personal := 5980
ELSE IF (age>=65)
     THEN personal := 5720
     ELSE personal := 4335
     FI
FI;
IF (age>=65 AND income >16800)
THEN IF (4335 > personal-((income-16800) / 2))
     THEN personal := 4335
     ELSE personal := personal-((income-16800) / 2)
     FI
FI;
IF (blind =1) THEN personal := personal + 1380 FI;
IF (married=1 AND age >=75)
THEN pc10 := 6692
ELSE IF (married=1 AND age >= 65)
     THEN pc10 := 6625
     ELSE IF (married=1 OR widow=1)
          THEN pc10 := 3470
          ELSE pc10 := 1500
          FI
     FI
FI;
IF (married=1 AND age >= 65 AND income > 16800)
THEN
   IF (3470 > pc10-(income-16800) / 2)
   THEN pc10 :=3470
   ELSE pc10 := pc10-((income-16800) / 2)
   FI
FI;
IF (income - personal <= 0)
THEN tax := 0
ELSE income := income - personal ;
FI;
IF (income <= pc10)
THEN tax := income * rate10
ELSE tax := pc10 * rate10 ;
     income := income - pc10 ;
FI;
IF (income <= 28000)
THEN tax := tax + income * rate23
ELSE tax := tax + 28000 *rate23 ;
     income := income - 28000 ;
     tax := tax + income * rate40
FI;
IF (blind=0 AND married=0 AND age<65)
  code := 'L';
ELSE IF(blind=0 age<65 AND married=1)
  code := 'H';
ELSE IF (age>=65 AND age<75 AND married=0 AND blind=0)
  code := 'P';
ELSE IF(age>=65 AND age<75 AND married=1 AND blind=0)
  code := 'V';
ELSE code := 'T';
FI
```

Conditioned program: boxed lines of code

Condition: `age`>=65 `AND age`<75 `AND income`=36000 `AND blind`=0 `AND married`=1

Figure 3.5: UK Income taxation calculation program in WSL

be reached along control flow paths under the given condition. However, the authors did not propose a formal definition of condition-based slicing. Field *et al.* [46] introduced the concept of constrained slice to indicate slices that can be computed with respect to any set of constraints. Their approach is based on an intermediate representation for imperative programs, named PIM, and exploits graph rewriting techniques based on dynamic dependence tracking [47] that model symbolic execution. The extracted slices are not executable. The authors were interested in the semantic aspects of more complex program transformations rather than in simple statement deletion.

An extension to conditioned slicing, namely backward conditioning, has been proposed by Danicic et al. [49]. While conditioned slicing uses forward conditioning, and deletes statements that are not executed when the initial state satisfies the condition, backward conditioning deletes statements which cannot cause execution to enter a state which satisfies the condition. Backward conditioning adresses questions of the form:

> ” What parts of the program could potentially lead to the program arriving in state satisfying a given condition ?”,

whereas forward conditioning adresses questions of the form:

> ”What happens if the program starts in a state satisfying a given condition ?”

Conditioned slicing has been applied to program comprehension [40, 49] and to the extraction of reusable functions [18]. The use of symbolic execution to specialise generalised software components to more specific and efficient functions to be used under more restricted conditions has been proposed by Coen-Porisini *et al.* [30].

3.3 Conditioned Slicing and Testing

When generating tests from a specification, it is common to apply partition analysis: a partition $P = \{D_1, \ldots, D_n\}$ of the input domain D, is produced. This partition has the property that the behaviour of the specification is uniform (and thus relatively simple) on each subdomain D_i. Faults may either affect the behaviour within a subdomain (computation faults) or affect the boundaries of the subdomains (domain faults).

Computation faults are detected by choosing one or more test cases from each subdomain. Domain faults are detected by testing around subdomain boundaries [27, 128]. Suppose an implementation under test I is tested on the basis of partition P. If I is *uniform* on each of the subdomains of P, it is likely that faults will be detected by a test set based on P. This form of assumption, that the behaviour is uniform on each D_i, is the 'uniformity hypothesis' of partition testing.

Conditioned slicing [17] is a technique for identifying those statements and predicates which contribute to the computation of a selected set of variables when some chosen condition is satisfied. The technique has previously been used in program comprehension [40, 49] and re-engineering [20]. Details about conditioned slicing are given in Section 3.2.

This section shows how conditioned slicing using the ConSUS slicing tool can be used to assist partition-based testing. Specifically it will be shown how conditioned slicing:

1. provides confidence in uniformity holding on a subdomain D_i from P;
2. suggests the existence of faults associated with subdomain $D_i \in P$, providing information that can be used to either refine P(domain faults) or direct effort towards

D_i(computation faults);

3. detects the existence of erroneous special cases.

These three topics are addressed by subsections 3.3.1, 3.3.2 and 3.3.3 respectively. All examples will be constructed with respect to the program in Figure 3.5, which calculates tax codes and tax rates for a United Kingdom citizen in the tax year April 1998 to April 1999.

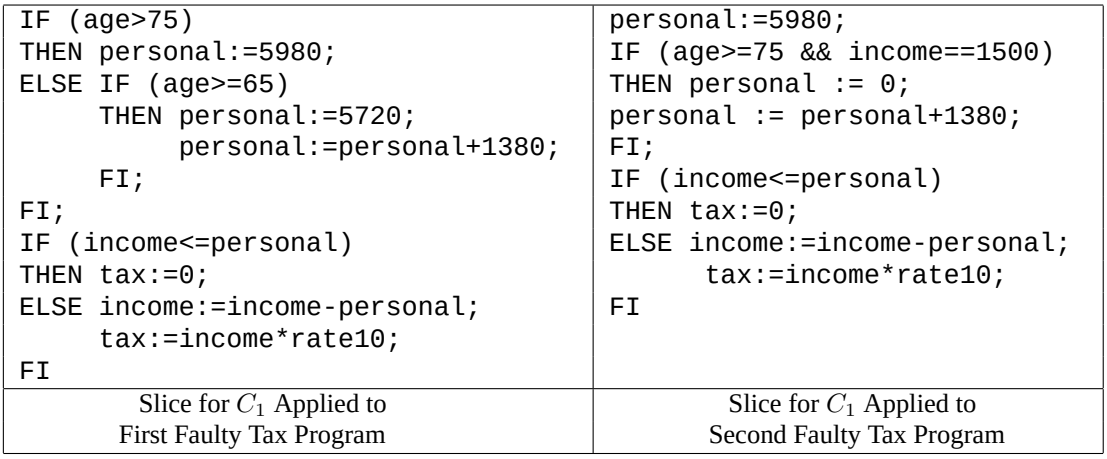
3.3.1 Fault Detection with Conditioned Slicing

One of the problems associated with partition analysis is that the behaviour of the implementation under test may not be uniform on each element of the partition. Where this assumption fails, the test generated on the basis of a prtition P is likely to be insufficient. It would therefore be useful to be able to determine whether the uniformity hypothesis holds. Where it does not hold for some $D_i \in P$, ideally the tester should either further divide D_i or choose more tests from D_i.

Let C_{D_i} denote the condition expressing the constraint that the input lies in D_i. Then, if I is uniform on D_i, the conditioned slice $S(I, C_{D_i})$ is likely to be relatively simple: slicing using condition C_{D_i} should lead to much simplification [61]. Where this is the case, the tester might have greater confidence in the uniformity hypothesis holding for D_i. Consider the tax example of Figure 3.5. Suppose the tester chooses the subdomain defined by the condition C_1 below:

$$age \geq 75 \; AND \;\; blind = 1 \; AND \;\; 0 \leq income \leq 7360$$

For this condition, and slicing on the variable `tax`, ConSUS produces the following con-



```
IF (age>75)
THEN personal:=5980;
ELSE IF (age>=65)
     THEN personal:=5720;
          personal:=personal+1380;
     FI;
FI;
IF (income<=personal)
THEN tax:=0;
ELSE income:=income-personal;
     tax:=income*rate10;
FI
```

Slice for C_1 Applied to
First Faulty Tax Program

```
personal:=5980;
IF (age>=75 && income==1500)
THEN personal := 0;
personal := personal+1380;
FI;
IF (income<=personal)
THEN tax:=0;
ELSE income:=income-personal;
      tax:=income*rate10;
FI
```

Slice for C_1 Applied to
Second Faulty Tax Program

Figure 3.6: Fault-revealing conditioned slices

ditioned slice.

```
tax := 0;
```

The simplicity of this conditioned slice suggests that the behaviour is uniform on this subdomain and thus that only a small number of tests are required here. Indeed, in this case, the slice is so simple that the tester can easily determine correctness.

3.3.2 Confidence Building with Conditioned Slicing

Suppose a fault is introduced by changing `IF(age >= 75)` to `IF(age > 75)`. ConSUS produces the slice in the left-hand column of Figure 3.6 for the subdomain defined by C_1. Here there has been far less simplification, suggesting that the behaviour may not be uniform. In particular, the conditioned slice contains `if` statements. In such situations, ConSUS can be of further assistance, by computing the simplest path conditions applicable. In this case it produces: $age = 75\ AND\ income <= 7100$, $age = 75\ AND\ income > 7100$, and $age > 75$.

`tax := 0;`	`personal := 5720;`
	`personal := personal + 1380;`
	`income := income - personal;`
	`tax := income*rate10;`
		`tax := 0;`

Slice for C_1^1 and variable `tax`	Slice for C_1^2 and variable `tax`	Slice for C_1^3 and variable `tax`

Figure 3.7: Conditioned slices for refined subdomains

This suggests that the subdomain denoted by C_1 should be refined to include each of the three path conditions, yielding:

1. $C_1^1 \equiv (C_1\ AND\ age = 75\ AND\ income <= 7100)$;
2. $C_1^2 \equiv (C_1\ AND\ age = 75\ AND\ income > 7100)$;
3. $C_1^3 \equiv (C_1\ AND\ age > 75)$.

For these refined domains, ConSUS produces the three slices in Figure 3.7. Values from the subdomain denoted by C_1^2 will detect the fault.

3.3.3 Highlighting Special Cases with Conditioned Slicing

Consider now a second fault, produced by adding the following extra (malicious) code just before the line that starts `IF(blind=1)`:

```
if (age >= 75 AND income = 1500) personal := 0;
```

Slicing using C_1 and variable `tax` yields the fragment in the right-hand column of Figure 3.6. This appears not to be uniform and thus the tester might either choose to test thoroughly within the corresponding subdomain, or to analyse the slice further. Further analysis of this slice leads to two new conditions:

1. $(income = 1500)$;

2. $NOT\ (income = 1500)$.

The fault will be found by refining the subdomain, corresponding to C_1, using these two conditions and then testing with samples from the refined domains.

Interestingly, this second fault is of a type that is usually very difficult to find using specification-based testing because the implementation contains behaviour that is *not* in the specification. Since the specification does not contain this behaviour, and the behaviour lies within the body of a subdomain, traditional specification-based testing is unlikely to find it: there is no information in the specification that indicates that the value 1500 for `income` is significant. Fortunately, conditioned slicing highlights this additional behaviour.

Chapter 4

The *ConSUS* Conditioning Algorithm

4.1 An Overview of the Approach

When implementing an interpreter, a program is evaluated in a state which maps variables to their values [110]. In symbolic execution [29, 34, 35, 52], the state, called a *symbolic store*[1], maps variables, not to *values*, but to *symbolic expressions* which may involve various uninterpreted values, constants and operators.

When a program is symbolically evaluated in an initial symbolic store, it gives rise to a collection of possible final symbolic stores. The reason that a symbolic evaluator returns a *collection* of final stores is that our program may have more than one path, each of which may define a different final symbolic store. Unlike the case of an interpreter, the initial symbolic store does not give rise to a unique path through the program. A *symbolic evaluator* can, thus, be thought of as a mapping, which given a program and a symbolic store, returns a collection of symbolic stores.

In order to implement a conditioner, a richer state space than that used in a symbolic evaluator is required. For each final symbolic store it is necessary also to record what

[1] Usually called the *symbolic state*.

properties must have been true of the initial symbolic store in order for the program to take the path that resulted in this final symbolic store. This is called a *path condition* and consists of a boolean expression involving constants and symbolic values.

A *conditioned state*, Σ, is represented by a set of path condition-symbolic store pairs. $\forall(b, \sigma) \in \Sigma$ then the symbolic store σ can be reached if path condition b is true. If a conditioned state contained the pair (false, σ), this would be equivalent to stating that the symbolic store σ is unreachable.

ConSUS can be thought of as a function which takes a program and an initial conditioned state and returns a (simplified) program and a final conditioned state[2]. In practice, a conditioner will normally be applied to programs starting in the *natural conditioned state*. In the natural conditioned state, the corresponding symbolic store, maps all variables to their names, representing the fact that no assignments have yet taken place. The corresponding path condition in the natural state is *true*, representing the fact that no paths have yet been taken.

4.1.1 Statement Removal

The program simplification produced by *ConSUS* arises from the fact that a statement from a program can be removed if all paths starting from the initial conditioned state of interest leading to the statement are infeasible. The path condition corresponding to a symbolic store is a condition which must be satisfied by the initial store in order for the program to take the path that arrives at the corresponding symbolic store. If the final path condition is equivalent to false then the store is not reachable.

[2]In [30], similar functions *exec* and *simpl* are defined. Fundamentally different, however, is that *exec* and *simpl* return a single path condition, symbolic state pair, not a set of such pairs as in our case.

The power of a conditioner, in essence, depends on the ability to prove that the path conditions encountered are tautologies or contradictions. This is why a conditioner needs to work in conjunction with a theorem prover. Of course, this is not a computable problem, infeasible paths may not be detected.

Consider again, the program in Figure 3.4. This program potentially has two possible final symbolic stores:

$$[a \rightarrow 1]$$

$$[a \rightarrow 2]$$

The corresponding path conditions are:

$$x > y \quad AND \quad y > z \quad AND \quad x > z$$

$$x > y \quad AND \quad y > z \quad AND \quad NOT(x > z).$$

Combining these two gives the conditioned state with two elements:

$$\{ \ (x > y \quad AND \quad y > z \quad AND \quad x > z, \quad [a \rightarrow 1]),$$

$$(x > y \quad AND \quad y > z \quad AND \quad NOT(x > z), \quad [a \rightarrow 2]) \ \}.$$

A sufficiently powerful theorem prover will be able to infer that the second of these path conditions is always false.

Often programs containing no assert statements will be conditioned. This corresponds to removing *dead* code. Consider the program in Figure 3.3. The programs in Figures 3.4 and 3.3 do not quite have the same semantics. The first will abort in initial stores not satisfying the initial path condition, while the second will do nothing but terminate successfully starting from these stores. The 'dead code' `a:=2` is removed by the conditioner

in both cases.

As will be shown later, *ConSUS* is efficient in the sense that it attempts to prune paths 'on the fly' as it symbolically executes. This is an improvement over some other systems like *ConSIT* [36] which generates all paths and then prunes once at the end. The way this is achieved is that on encountering a guard, *ConSUS* interacts with its theorem proving mechanism to check whether the negation of the symbolic value of the guard is implied by the corresponding path condition in all values of the current conditioned state. If this is the case, then the corresponding body is unreachable and so can be removed without being processed.

Programs containing loops may have infinitely many paths. These cannot all be considered and therefore a conservative and safe approach has to be adopted when conditioning loops. For each `WHILE` loop, it is essential that in any implementation only a finite number of distinct symbolic stores are generated. A *meta symbolic store* is required in order to represent the infinite set of symbolic stores that are not distinguished between. This *meta symbolic store* must be safe in the sense that it must not add any untrue information about these symbolic stores. The simplest possible approach is simply to 'throw away' any information about variables which are affected by the body of a loop. This idea is very similar to state folding introduced in [30]. Their program specialiser, returns a single symbolic store, path condition pair, and so it is necessary to throw away values corresponding to variables assigned different values on each branch of an `IF THEN ELSE` statement.

Using this approach, a `WHILE` loop will map each symbolic store, σ, to a set consisting of two symbolic stores. One of the stores will be σ itself, (representing the fact that that the guard of the loop may be initially *false*) and the other store (representing the fact that

```
x:=y+1;
WHILE x>y
DO
   x:=y+2
OD;
IF x=y
THEN p:=7;
```

Key

Code removed using the naïve approach: None
Code removed using the *ConSIT* approach: boxed lines of code

Figure 4.1: Conditioning a `WHILE` loop using two approaches

the loop was executed at least once) will be represented by a store, σ', which agrees with σ on all variables not affected by the body of the loop. In σ', all variables that *are* affected by the body of the loop are *skolemised*, representing the fact that we no longer have any information about their value. By skolemising a variable, all previous information that we had about it is being thrown away. As a result of skolemising a symbolic store, incorrect information will never be generated, it will just be less precise.

The approach taken by *ConSUS* (based on the approach of *ConSIT* [36]) is less crude, however. In this case, symbolically evaluating a `WHILE` loop, results in the set consisting of σ as before, together with the set of stores which are the result of symbolically executing the body of the loop in the skolemised store σ'. To see how the two approaches differ, consider the example given in Figure 4.1. Using the naïve approach, the two symbolic stores resulting from the `WHILE` loop are $[x \rightarrow y+1]$ and $[x \rightarrow x_0]$. The first of these represents not executing the loop at all and the second represents the fact that the loop body has been executed at least once. The variable x has been skolemised to x_0, representing the fact that its value is no longer known. Evaluating the guard $x = y$ of the `IF-THEN` statement in this skolemised store gives $x_0 = y$. Since $x_0 = y$ is not a contradiction, the conditioner using the naïve approach would be forced to keep in the whole `IF-THEN`

statement, however powerful the theorem prover.

Using the less crude approach gives the two symbolic stores $[x \rightarrow y+1]$ and $[x \rightarrow y+2]$. The fact that in the loop, x is assigned an expression that is unaffected by the body of the loop has been taken into account. Since $y+1=y$ and $y+2=y$ are both contradictions, the `IF` statement following the `WHILE` loop can be removed.

4.2 The *ConSUS* Algorithm in Detail

In this section, the algorithm used by *ConSUS* is explained in detail. For each WSL syntactic category, the result of applying *ConSUS* to it will be defined. It will be assumed that the starting conditioned state in each case is given by:

$$\Sigma = \bigcup_{i=1}^{n} \{(b_i, \sigma_i)\}$$

where the b_i are boolean expressions representing path conditions and the σ_i are the corresponding symbolic stores.

For each statement s, *ConSUS* returns two objects:

- *state*(Σ,s): the resulting conditioned state when conditioning statement s in Σ and
- *statement*(Σ,s): the resulting simplified statement when conditioning statement s in Σ.

  If statement s is to be removed by *ConSUS*, it returns `SKIP`. A final post-processing phase will call *FermaT's Delete_All_Skips* transformation to remove all the `SKIP`s

that have introduced by performing this operation.

Calls to the theorem prover, *FermaT* `Simplify` will be represented by the expression $prove(b)$, where b is a boolean expression. The expression, $prove(b)$, is defined to return *true* if the theorem prover determines that b is valid and *false* otherwise. If $prove(b)$ returns *false*, this represents the fact that *either* the theorem prover cannot reduce the condition to *true or* it reduces it to the condition to *false*.

Given a conditioned state, Σ, and a boolean expression b, we define *AllImply*(Σ, b) to be *true* if and only if, for all pairs (c, σ) in conditioned state Σ, *prove*$(c \implies \sigma\ b)$ evaluates to *true*. Where, given a symbolic store σ, the expression, $\sigma\ b$, denotes the result of symbolically evaluating b in σ.

Suppose b is the guard of an `IF` statement. *AllImply*(Σ, b) implies that the `THEN` branch *must* be executed in Σ and the `ELSE` branch can be removed. Similarly *AllImply*$(\Sigma,$ `NOT` $b)$ implies that `THEN` branch can be removed. Suppose b is a guard of a `WHILE` loop, then *AllImply*(Σ, b) implies that the body of the loop is executed at least once and *AllImply*$(\Sigma,$ `NOT` $b)$ implies that the loop body is not executed at all.

4.2.1 Conditioning `ABORT`

In order to condition an `ABORT` statement, a special conditioned state called the `ABORT` state is introduced and written $\perp$. It consists of the single pair $(\mathit{false}, \mathit{id})$.

$$\mathit{state}(\Sigma, \texttt{ABORT}) \triangleq \perp$$

$$\mathit{statement}(\Sigma, \texttt{ABORT}) \triangleq \texttt{ABORT}$$

For all statements s, define

$$\textit{state}(\bot, s) \triangleq \bot$$

$$\textit{statement}(\bot, s) \triangleq \texttt{SKIP}$$

This guarantees that all statements following an ABORT will be removed. In the rest of the discussion it is assumed that $\Sigma \neq \bot$.

4.2.2 Conditioning SKIP

$$\textit{state}(\Sigma, \texttt{SKIP}) \triangleq \Sigma$$

$$\textit{statement}(\Sigma, \texttt{SKIP}) \triangleq \texttt{SKIP}$$

Conditioning a SKIP has no effect.

4.2.3 Conditioning Assert Statements

In WSL, an assert statement is written $\{b\}$ where b is a boolean expression. It is semantically equivalent to IF b THEN SKIP ELSE ABORT FI. There are three cases to consider:

Case	Condition	Meaning
1	*AllImply*(Σ, b)	The assert condition will always be *true*
2	*AllImply*$(\Sigma,$ NOT $b)$	The assert condition will always be *false*
3	None of the above	Nothing can be inferred

From the semantics of the Assert statement it is clear that in case 1, the Assert is equivalent to SKIP so the rules for SKIP above apply. In case 2, the Assert is equivalent to ABORT so the rules for ABORT above apply. If neither the guard of the Assert is not

always *true* or not always *false* in the current state, then the Assert cannot be removed. The resulting state will have the same set of symbolic stores.

The path conditions of the resulting state will be different however. For each pair, (b_i, σ_i) the resulting state will have a corresponding pair $(b_i \text{ AND } \sigma_i\ b, \sigma_i)$ where b_i AND $\sigma_i b$ is the boolean expression created by conjoining the boolean expression b_i with the result of symbolically evaluating the boolean expression[3] b in symbolic store σ_i. This represents the fact that a program will continue executing after an Assert statement in stores where b evaluates to *true*. Formally, in this case,

$$\textit{state}(\Sigma, \{b\}) \triangleq \bigcup_{i=1}^{n} \{(b_i \text{ AND } \sigma_i b,\ \sigma_i)\}.$$

$$\textit{statement}(\Sigma, \{b\}) \triangleq \{b\}.$$

4.2.4 Conditioning Assignment Statements

When conditioning assignment statements, *ConSUS* symbolically evaluates the expression on the right hand side of the assignment and updates the symbolic stores accordingly. The path conditions do not change. In order to symbolically evaluate an expression e in a symbolic store, σ, *ConSUS* replaces every variable in the expression by its value in σ. Given a symbolic store, σ, we use standard notation $\sigma[x \rightarrow e]$ to represent a store that 'agrees' with σ except that variable x is now mapped to e. Using this, the conditioning of assignment statements can be defined as follows:

$$\textit{state}(\Sigma, x := e) \triangleq \bigcup_{i=1}^{n} \{\ (b, \sigma_i[x \rightarrow \sigma_i e])\ \}$$

$$\textit{statement}(\Sigma, x := e) \triangleq x := e.$$

[3]For example, if σ_i maps y to $z+1$ and x to 17 and if b is the boolean expression: $y > x+1$ and if b_i is the boolean expression: $a + z = 5$ then $(b_i \text{ AND } \sigma_i b)$ is the boolean expression: $a + z = 5$ AND $z + 1 > 17 + 1$.

4.2.5 Conditioning Statement Sequences

In the case of standard semantics [110], the meaning of a sequence of statements is the composition of the meaning functions of the individual statements. The same is true when conditioning:

$$\mathit{state}(\Sigma, s_1;\ s_2) \triangleq \mathit{state}(\mathit{state}(\Sigma, s_1), s_2)$$

$$\mathit{statement}(\Sigma, s_1;\ s_2) \triangleq \mathit{statement}(\Sigma, s_1);\ \mathit{statement}(\mathit{state}(\Sigma, s_1), s_2)$$

This reflects the fact that conditioned states are 'passed through' the program in the same order that the program would have been executed. Once again, if as a result of conditioning, both parts of the sequence reduce to `SKIP` then they will both be removed by the post-processing phase.

4.2.6 Conditioning Guarded Commands

In WSL, a generalised form of conditional known as guarded command is used. A guarded command has concrete syntax of the form

$$\texttt{IF}\ B_1\ \texttt{THEN}\ S_1\ \texttt{ELSIF} \cdots \texttt{ELSIF}\ B_n\ \texttt{THEN}\ S_n\ \texttt{FI}.$$

Unlike the semantics of Dijkstra's guarded commands [42], these are deterministic in the sense that the guards are evaluated from left to right and when a true one is found the corresponding body is executed. If none of the guards evaluates to *true* then the program aborts. Although WSL has conventional `IF THEN ELSE FI` statement, these are implemented as a guarded command whose last guard is identically `TRUE`. An `IF THEN` statement is also implemented as a guarded command whose last guard is identically `TRUE` and whose corresponding body is `SKIP`. For the purposes of describing condition-

ing guarded commands, it is convenient to represent a guarded command as

$$B_1 \rightarrow S_1 | \ldots | B_n \rightarrow S_n.$$

Using WSL terminology, each $B_i \rightarrow S_i$ is known as a *guarded*. Conditioning a guarded command is defined in terms of conditioning a guarded, $B \rightarrow S$ so that is defined first.

When conditioning a guarded, like in the case of the Assert statement, there are three possibilities:

Case	Condition	Meaning
1	*AllImply*(Σ, B)	The guard B will always be *true*
2	*AllImply*$(\Sigma, \texttt{NOT}\ B)$	The guard B will always be *false*
3	None of the above	Nothing can be inferred

In cases 1 and 3,

$$\textit{state}(\Sigma, B \rightarrow S) \triangleq \textit{state}(\Sigma', S)$$

$$\textit{statement}(\Sigma, B \rightarrow S) \triangleq B \rightarrow \textit{statement}(\Sigma', S)$$

where

$$\Sigma' = \bigcup_{i=1}^{n} \{(b_i\ \texttt{AND}\ \sigma_i B,\ \sigma_i)\}.$$

In case 2, the guarded can be removed and the resulting state will simply be Σ:

$$\textit{state}(\Sigma, B \rightarrow S) \triangleq \Sigma$$

$$\textit{statement}(\Sigma, B \rightarrow S) \triangleq \texttt{SKIP}$$

Having defined how *ConSUS* conditions a single guarded, we now return to define how *ConSUS* conditions a complete guarded command. As already explained, a guarded command is a sequence of guardeds:

$$B_1 \rightarrow S_1 | \ldots | B_n \rightarrow S_n.$$

When conditioning a guarded command in Σ, the guardeds are conditioned, as described above, from left to right. The jth guarded is conditioned in conditioned state Σ_j where

$$\Sigma_1 = \Sigma$$

and

$$\Sigma_{j+1} = \bigcup_{(b_i, \sigma_i) \in \Sigma_j} \{(b_i \texttt{ AND } \sigma_i \texttt{ NOT } B_j, \sigma_i)\}.$$

For each guarded, $B_j \rightarrow S_j$, *ConSUS* decides:

(a) Whether to keep or remove it.

(b) Whether to continue processing the next guarded in this guarded command or to move on to the next statement after the guarded command.

Conditioning proceeds as follows:

- If *AllImply*(Σ_j, B_j) this implies that the jth guard will be chosen in all paths where the previous guards have not been chosen. The resulting statement will be *statement*$(\Sigma_j, B_j \rightarrow S_j)$. Conditioning of the guarded command can stop at this point since none of the guardeds to the right of this one will ever be executed in Σ.

- If *AllImply*$(\Sigma_j, \texttt{NOT}\ B_j)$ this implies that the jth guard will never be chosen. This guarded can, therefore, be removed without conditioning it, and processing can continue with the conditioning of the next guarded, $B_{j+1} \rightarrow S_{j+1}$ in conditioned state $\Sigma_{j+1} = \Sigma_j$.

- If neither *AllImply*(Σ_j, B_j) nor *AllImply*$(\Sigma_j, \texttt{NOT}\ B_j)$ then it cannot said for certain whether B_j will be chosen or not. This is represented by keeping the guarded, *statement*$(\Sigma_j, B_j \rightarrow S_j)$, and again moving on to process the next guarded in conditioned state Σ_{j+1}.

Processing continues in this way from left to right until there are no more guardeds to consider. The resulting final conditioned state of the guarded command is the union of all the conditioned states of the guardeds that were processed. The resulting final statement of the guarded command is either:

1. a guarded command consisting of the guardeds that were kept in by the above process, in the same order (This rule only applies if more than one guarded was kept in by the above process.) or

2. the body of the only guarded that was kept in. (This rule only applies if exactly one guarded was kept in by the above process.) or

3. ABORT (this rule only applies if no guardeds were kept in by the above process.)

Since, as described above, not all guardeds need necessarily be processed, this algorithm is, in effect, pruning infeasible paths ‘on the fly’. This is a much more efficient approach than that of *ConSIT* [36], where all paths were fully expanded before any simplification took place.

4.2.7 Conditioning Loops

Before the result of conditioning `WHILE` B `DO` S `OD`, in conditioned state Σ is defined, some preliminary definitions are required.

Definition 1*:* Σ^{true} is the initial state Σ with the added constraint that the guard, B, is initially true in all pairs of Σ.

$$\Sigma^{true} = \bigcup_{(b,\sigma)\in\Sigma} \{(b \texttt{ AND } (\sigma B), \sigma)\}.$$

Similarly,

Definition 2*:* Σ^{false} is the initial state Σ with the added constraint that the guard, B, is initially false in all pairs of Σ.

$$\Sigma^{false} = \bigcup_{(b,\sigma)\in\Sigma} \{(b \texttt{ AND } (\sigma \texttt{ NOT } B), \sigma)\}.$$

Definition 3 *(The Skolemised Conditioned State,* Σ'*):* The skolemised conditioned state

$$\Sigma' = \bigcup_{(b,\sigma)\in\Sigma^{true}} \{(b, \sigma')\}.$$

where the symbolic stores, σ'_i, are the skolemised versions of the σ_i with respect to S, as described in Subsection 4.1.

Definition 4 *(*$\Sigma^{\geq 1}$*):* $\Sigma^{\geq 1}$ is the conditioned state after at least one execution of loop in state Σ.

$$\Sigma^{\geq 1} = \mathit{state}(\Sigma', S).$$

where the symbolic stores, σ'_i, are the skolemised versions of the σ_i with respect to S.

Definition 5 *(*Σ^{final}*):* Σ^{final} is the final conditioned state after at least one execution of the

	AllImply$(\Sigma, \texttt{NOT}\ B)$	*AllImply*(Σ, B)	*AllImply*$(\Sigma^{\geq 1}, \texttt{NOT}\ B)$	*AllImply*$(\Sigma^{\geq 1}, B)$
Case 1	T			
Case 2	F	F	F	F
Case 3	F	F	F	T
Case 4	F	F	T	F
Case 5	F	T	F	F
Case 6	F	T	F	T
Case 7	F	T	T	F

Figure 4.2: `WHILE` loop possibilities

loop in state Σ assuming that the loop terminates.

$$\Sigma^{\text{final}} = \bigcup_{(b,\sigma)\in\Sigma^{\geq 1}} \{(b\ \texttt{AND}\ \sigma(\texttt{NOT}\ B), \sigma)\}.$$

When conditioning a loop of the form `WHILE` B `DO` S `OD`, in conditioned state Σ, *ConSUS* checks all the seven conditions in the table in Figure 4.2.

Each case in Figure 4.2 has the following implications:

Case 1	Loop not executed
Case 2	Nothing known
Case 3	If loop executed once, then it does not terminate
Case 4	If loop executed once, then it executes exactly once
Case 5	Loop executes at least once
Case 6	Loop non-terminates
Case 7	Loop executes exactly once

Blank entries in the table mean we do not care about these values. The other combinations not considered are all impossible. For each of these cases,

	Final State
Case 1 (Loop not executed)	Σ
Case 2 (Nothing known)	$\Sigma^{\mathit{false}} \cup \Sigma^{\text{final}}$
Case 3 (If once, non-termination)	Σ^{false}
Case 4 (If once, exactly once)	$\mathit{state}(\Sigma, \texttt{IF}\ B\ \texttt{THEN}\ S\ \texttt{FI})$
Case 5 (At least once)	Σ^{final}
Case 6 (Non-termination)	$\bot$
Case 7 (Exactly once)	$\mathit{state}(\Sigma, S)$

Figure 4.3: `WHILE` loop final states in each case

	Final Statement
Case 1 (Loop not executed)	`SKIP`
Case 2 (Nothing known)	`WHILE` B `DO` $\mathit{statement}(\Sigma', S)$ `OD`
Case 3 (If once, non-termination)	{ `NOT` B }
Case 4 (If once, exactly once)	$\mathit{statement}(\Sigma, \texttt{IF}\ B\ \texttt{THEN}\ S\ \texttt{FI})$
Case 5 (At least once)	`WHILE` B `DO` $\mathit{statement}(\Sigma', S)$ `OD`
Case 6 (Non-termination)	`ABORT`
Case 7 (Exactly once)	$\mathit{statement}(\Sigma, S)$

Figure 4.4: `WHILE` loop resulting statements in each case

$$\mathit{state}(\Sigma, \texttt{WHILE}\ B\ \texttt{DO}\ \ S\ \ \texttt{OD})$$

and

$$\mathit{statement}(\Sigma, \texttt{WHILE}\ B\ \texttt{DO}\ \ S\ \ \texttt{OD})$$

will have different values (Figures 4.3 and 4.4). Each is now considered in turn.

Case 1: the loop is not executed. There is no change to the final conditioned state and loop can be removed.

Case 2: nothing is known about the loop. The final conditioned state is the union of the final conditioned states corresponding to not executing the loop at all and to terminating after at least one execution. It is not necessary to consider non-termination as no states after non-termination are reachable. The resulting statement is the while loop with its body conditioned in Σ', where Σ' is the skolemised state.

Case 3: if the loop is executed at least once then it fails to terminate. The final conditioned state corresponds to not executing the loop, since this is the only way termination can occur. The loop can be replaced with an assertion of the negation of the guard.

Case 4: if the loop is executed once then it executes at most once. This is equivalent to conditioning the corresponding conditional statement in state Σ.

Case 5: the loop is executed at least once. The final conditioned state is the Σ^{final}, corresponding to the loop terminating after at least one execution. It is not necessary to consider non-termination as no states after non-termination are reachable. The resulting statement is the while loop with its body conditioned in skolemised state, Σ'.

Case 6: the loop does not terminate. The final state is $\bot$ and the loop can be replaced with ABORT.

Case 7: The loop executes exactly once. This is equivalent to conditioning S in Σ. Since *AllImply*(Σ, B) and *AllImply*$(\Sigma^{\geq 1}, \texttt{NOT}\ B)$ we do not need to add the constraints that the loop guard is initially *true* and finally *false*.

4.3 Examples

This section gives examples of the output of *ConSUS* for a variety of small examples in order to demonstrate its behaviour.

The program in Figure 4.5 is an example with two consecutive identical while loops. *ConSUS* removes the second loop since its guard can never be true after completing execution of the first loop. This is true even if the first loop is not executed or if it non-terminates.

`WHILE x<1`
`DO x:=x+1`
`OD;`
`WHILE x<1`
`DO x:=x+1`
`OD`
	`WHILE x<1`
	`DO x:=x+1`
	`OD`

Original Program	Output from *ConSUS*

Figure 4.5: Conditioning a `WHILE` loop (Case 1)

`x:=p;`
`WHILE x>0`
`DO x:=1`
`OD;`
`IF x=p`
`THEN y:=2`
`ELSE y:=1`
`FI`
	`x:=p;`
	`{NOT x > 0};`
	
	`y :=2`

Original Program	Output from *ConSUS*

Figure 4.6: Conditioning a `WHILE` loop (Case 3)

`WHILE x=1`
`DO x:=2`
`OD`
	`IF x=1`
	`THEN x:=2`
	`FI`

Original Program	Output from *ConSUS*

Figure 4.7: Conditioning a `WHILE` loop (Case 4)

`x:=1;`
`WHILE x>0`
`DO x:=x+y;`
`  y:=2`
`OD;`
`IF (y=2)`
`THEN x:=1`
`ELSE x:=2`
`FI`
	`x:=1;`
	`WHILE x>0`
	`DO x:=x+y;`
	`  y:=2`
	`OD;`
	
	`x:=1`

Original Program	Output from *ConSUS*

Figure 4.8: Conditioning a `WHILE` loop (Case 5)

In Figure 4.6 there is a loop which if, executed once, never terminates. *ConSUS* replaces this loop with an Assert statement that asserts that the guard of the loop is false. *ConSUS* also recognised that to 'get past' the loop, it must not be executed and therefore the initial assignment to x is not overwritten, and so the following IF statement can be simplified.

The program in Figure 4.7 has a while loop which is executed exactly once or not at all. *ConSUS* replaces it with an IF statement. In the current implementation, if the `2` was replaced by `x+1`, say, no simplification would take place. This is because the *ConSUS* infers that only a single loop iteration is possible, by analysing the loop guard in the skolemised state and not in the state after a single execution.

In Figure 4.8, although the loop itself cannot be simplified, *ConSUS* recognises that the

`{x>1};`
`WHILE x>0`
`DO y:=x+y;`
`OD;`
`IF (x>0)`
`THEN x:=1`
`ELSE x:=2`
`FI`
	`{x>1};`

Original Program	Output from *ConSUS*

Figure 4.9: Conditioning a `WHILE` loop (Case 6)

`x=1;`
`WHILE x=1`
`DO x:=x+1;{x>1}`
`OD;`
`IF (x=1)`
`THEN x:=1`
`ELSE x:=2`
`FI`
	`x;=1`
	`x:=x+1;{x>1}`
	`x:=2`

Original Program	Output from *ConSUS*

Figure 4.10: Conditioning a `WHILE` loop (Case 7)

loop must be executed at least once and hence the later IF statement can be simplified.

In Figure 4.9, *ConSUS* recognises that the program does not terminate and therefore everything apart from the initial Assert can be discarded since these statements are not reachable.

In Figure 4.10 we have ‘helped’ the theorem prover with the knowledge that in this loop, `x` will always be greater than zero. From this, *ConSUS* has inferred that the loop will terminate after exactly one execution. As the implementation stands, without this human intervention, *ConSUS* would not produce any simplification. As in Case 4, this is because

the *ConSUS* infers that only a single loop iteration is possible, by analysing the loop guard in the skolemised state, and not in the state after a single execution. The algorithm could very straightforwardly be changed to consider one iteration of the loop as a special case. In this example, we see that slicing on `x` at the end of the program before conditioning yields no simplification. But after conditioning, slicing on `x` at the end of the program gives us the single statement `x:= 2`.

Chapter 5

Design and Implementation

5.1 Architecture of the System

The system was designed and implemented in a LINUX environment. It was considered appropriate to design the system in a modular fashion, so that two programmers could independently solve their parts of the problem. The final phase would consist of the integration of these independent parts. The two parts are:

1. The Conditioner
2. The Slicer

The overall architecture is presented in 5.1.

Figure 5.1: Top-level Architecture

For illustartive purposes, an example of a program that the system accepts is given in 5.2. This program codifies part of the UK tax system for the year 1999–2000.

The two annotations give the conditioning and slicing criteria. They state that we are only interested in the part of the computation that is relevant for blind widows under 50

```
  { blind = 1 AND widow = 1 AND age < 50 };
IF (age>=75)
THEN personal := 5980;
ELSE IF (age>=65)
     THEN personal := 5720;
     ELSE  personal := 4335;
     FI;
FI;
IF ((age>=65) AND income>16800)
THEN t := personal - ((income-16800)/2);
     IF (t>4335)
     THEN personal := t;
     ELSE personal := 4335;}
     FI; FI;
IF (blind=1)
THEN  personal := personal + 1380;
IF (married=1 AND age>=75)
THEN pc10 := 6692;
ELSE IF (married=1 AND (age>=65))
     THEN pc10 := 6625;
     ELSE IF (married=1 OR widow=1)
          THEN  pc10 := 3470;
          ELSE pc10 := 1500;
          FI;
     FI;
FI;
IF (married = 1 AND age>=65 AND income>16800)
THEN t := pc10-((income-16800)/2);
     IF (t>3470)
     THEN pc10 := t;
     ELSE pc10 = 3470;
     FI;
FI;
```

```
IF (income<=personal)
THEN tax := 0;
ELSE income := income-personal;
     IF(income<=pc10)
     THEN tax := income/10;
     ELSE tax := pc10/10;
          income := income-pc10;
          IF (income<=28000)
          THEN tax := ((tax+income)*23)/100;
          ELSE tax := ((tax+28000)*23)/100;
               income := income-28000;
               tax := ((tax+income)*40)/100;
          FI;
     FI;
FI;
IF (blind=0 AND married=0 AND age<65)
THEN code = 'L';
ELSE IF (blind=0 AND age<65 AND married=1)
     THEN code = 'H';
     ELSE IF (age>=65 AND age<75 AND married=0 AND blind=0)
          THEN code = 'P';
          ELSE IF (age>=65 AND age<75 AND married=1 AND blind=0)
               THEN code = 'V';
               ELSE code = 'T';
               FI;
          FI;
     FI;
FI;
 slice := tax;
}
```

Figure 5.2: Tax for Blind Widow Under 50

years old, and that we are only interested in the value of tax, as computed at the end of the program. The annotations are part of the program that are accepted by the system.

On processing this program (with respect to the slicing criterion), the slicer will remove the code in gray, as it is not relevant to the computation of `tax`, effectively leaving just remaining code in shadowed boxes , which the conditioner has found to be relevant under the given conditions, namely that we are only interested in the computation for blind widows under 50 years old.

In effect, both slicing and conditioning are techniques for isolating interesting parts of a program. Combining them into conditioned slicing produces a system which can be used to help isolate the parts of a program that are involved in computing the values of specified variables at specified points, under given execution conditions.

The implementation of our slicing and conditioning algorithms were achieved using a *WSL* [91]. *WSL* is both the language that the slicer and the conditioner were written in as well as the object language to be conditioned sliced. The reason for our choice is that *WSL* has a built in *WSL* parser that can be called from within a *WSL* program as well as a whole transformation system which is useful for the simplification of slices. Transformations were not a major design criterion of most popular programming languages. However, *WSL* (wide spectrum language) and transformation theory form the basis of the 'Maintainer's Assistant'tool [123] used for analysing programs by transformations. *WSL* is also the basis of the FermaT transformation system. The FermaT transformation system applies correctness-preserving transformations to programs written in *WSL* language. It is an industrial-strength engine with many applications in program comprehension and language migration, it has been used in migration IBM assembler to C and to COBOL [90]. Low-level programming constructs and high-level abstract specifications are both included in *WSL* language; hence the transformation of a program from abstraction specification to a detailed implementation can be expressed in a single language. The syntax and semantics of *WSL* are described in [91].

5.1.1 The Slicer

In order to slice a program P with respect to a set of variables V, we define a function called $Slice(P,\ V)$ which takes as arguments the program to be sliced and a set of variables to slice with respect to and returns the resulting slice. we also define a function called $Needed(P,\ V)$ which take a program P and a set of Variables V and returns a new set of variables produced by Slicing P with respect to V.

Assignment Statement

(S) $\qquad\qquad$ $x\ :=\ e$

Slicing the assignment statement $(x := e)$ with respect to a set of variables V is achieved as follows:

$$Slice(x := e,\ V) = \begin{cases} Skip & if\ x \notin \mathrm{V} \\ x := e & if\ x \in \mathrm{V} \end{cases}$$

$$Needed(x := e,\ V) = \begin{cases} V & if\ x \notin V \\ (V\ /\ x) \ \cup\ Variables(e) & if\ x \in \mathrm{V} \end{cases}$$

where Variables(e) is the reference variables in the assignment $x := e$.

Sequence of Statements

$$S_1;\ S_2;$$

To slice the sequence of statements $(S_1; S_2)$, with respect to a set of variables V we start by slicing S_2 with respect to V. This produces a new slice R_2 and a new set of variables V_2, we then slice S_1 with respect to V_2 producing R_1 and V_1.

$$\begin{aligned} Slice(S_2, V) &= R_2 \\ Needed(S_2,\ V) &= V_2 \\ Slice(S_1, V_2) &= R_1 \\ Needed(S_1,\ V_2) &= V_1 \end{aligned}$$

$$Slice(S_1;\ S_2,\ V) = \begin{cases} Skip & if\ \ R_1 = R_2 = Skip \\ R_1;\ \ R_2 & if\ \ R_1 \vee\ R_2 \neq\ Skip \end{cases}$$

$$Needed(S_1;\ S_2,\ V) = \begin{cases} V & if\ \ R_1 = R_2 = Skip \\ V_1 & if\ \ R_1 \vee\ R_2\ \neq\ Skip \end{cases}$$

IF Statement

$$(S) \qquad if \quad B \quad then \quad S_1 \quad else \quad S_2 \quad fi\ ;$$

In order to slice the above IF statement (S) with respect to a set of variables V we slice both the true part S_1 and the else part S_2 separately with respect to V.

$$\begin{aligned} Slice(S_1, V) &= R_1 \\ Needed(S_1,\ V) &= V_1 \end{aligned}$$

$$\begin{aligned} Slice(S_2, V) &= R_2 \\ Needed(S_2,\ V) &= V_2 \end{aligned}$$

$$Slice(S,\ V) = \begin{cases} Skip & if\ \ R_1 = R_2 = Skip \\ if\ \ B\ then\ R_1\ \ else\ \ R_2\ fi & if\ \ R_1 \vee\ R_2\ \neq\ Skip \end{cases}$$

$$Needed(S,\ V) = \begin{cases} V & if\ \ R_1 = R_2 = Skip \\ V_1\ \cup\ V_2\ \cup Variables(B) & if\ \ R_1 \vee\ R_2\ \neq\ Skip \end{cases}$$

If both slices R_1 and R_2 skip then the IF statement has no effect on any variable of interest, therefore it is replaced with skip and the $Needed$ new set of variables

is unchanged and is equal to V. Otherwise, the slice yields the following statement ($if \quad B \quad then \quad R_1 \quad else \quad R_2 \quad fi$). And the $Needed$ new set of variables becomes $V_s = V_1 \cup V_2 \cup Variables(B)$.

While Statement

$$(S) \qquad\qquad while \quad B \quad do \quad S_1 \quad od$$

In order to slice the above while statement with respect to a set of variables V we take the following steps: first we slice the body of S of the while statement with respect to V:

$$Slice(S_1,\ V) = R_1$$

If R_1 = $Skip$
In this case the whole while statement is replaced with $skip$ and the new set of variables is unchanged and is equal to V.

$$\begin{aligned} Slice(S,\ V) &= Skip \\ Needed(S,\ V) &= V \end{aligned}$$

If $R_1 \neq Skip$ then let:

$Needed_0$ = $Needed\ (if\ B\ then\ S_1\ else\ Skip\ fi,\ V)$

$Needed_{i+1}$ =

$Needed\ (S_1,\ Needed_i \cup V) \cup Needed_i$

Let n be the first integer such that $Needed_{n+1} \subset Needed_n$ then:

$$Slice(S,V) = while\ B\ do\ Slice\ (S_1,\ Needed_n\ \cup\ V)\ od$$
$$Needed(S,\ V) = Needed_n$$

$$Slice(S,\ V) = \begin{cases} Skip & if\ \ R_1 = Skip \\ while\ B\ do\ Slice\ (S_1,\ Needed_n\ \cup\ V)\ od & if\ \ R_1 \neq\ Skip \end{cases}$$

$$Needed(S,\ V) = \begin{cases} V & if\ \ R_1 = Skip \\ Needed_n & if\ \ R_1\ \neq Skip \end{cases}$$

5.2 Implementation Details

Details of the WSL implementation of *ConSUS* are now provided. In particular we concentrate on aspects of the code which demonstrate how WSL facilitates the implementation. The basic data structure for conditioned states is a list of symbolic store, path condition pairs as described in Chapter 4. A store is represented as a list of variable name, expression pairs. Expressions and all other syntactic components are stored using `MetaWSL`'s internal representation thus enabling them to be accessed and constructed naturally in `MetaWSL`. A good example is the function `@Subst`, given in Figure 5.3, which evaluates an expression in a symbolic store. This function is now explained in detail. The line

```
MW_FUNCT @Subst(store,exp) ==
```

is a function heading. The name of the function is `@Subst`. `@Subst` has two formal parameters `store` and `exp`. (Note, variables are not typed in WSL).

```
MW_FUNCT @Subst(store,exp) ==
VAR <R := < > > :
@Edit;
@New_Program(exp);
FOREACH Variable
DO
   IF @N_String(@V(@I)) IN @Domain(store)
   THEN @Paste_Over( @ValueOf(store,@N_String(@V(@I))))
   FI;
OD;
R := @I;
@Undo_Edit;
( R ) . ;
```

Figure 5.3: Evaluating an expression in a symbolic store

```
VAR <R := < > > :
```

Is a declaration of a local variable R, whose initial value is the empty list. Unfortunately, in WSL, local variables cannot be declared without giving them an initial value. The structure,

```
@Edit;
@New_Program(exp);
...
@Undo_Edit;
```

is typical in `MetaWSL`, the `@Edit` command has the effect of putting the current special global variable, `@I` on the stack and temporarily assigning `@I`, to the syntax tree corresponding to the expression `exp`. The `@Undo_Edit` command pops of the previously stacked value into `@I`. This useful technique can be used for all elements of abstract syntax such as expressions, statements, sequences of statements etc.

The `foreach` construct is an example of a high-level construct in `MetaWSL`. A `foreach` is used to iterate over all those components of the currently selected syntac-

tic item which satisfy certain conditions, and apply various editing operations to them. Within the body of the `foreach` it appears as if the current syntactic item is the whole program. The construct takes care of all the details, when for example, components are deleted, expanded or otherwise edited. The code fragment in Figure 5.3

```
@Edit;
@New_Program(exp);
FOREACH Variable
DO
...
OD;
R := @I;
@Undo_Edit;
```

will, thus, have the effect of repeating the action between the `DO` and `OD` for each variable in the expression `exp` and storing the resulting transformed in `R`. This code does not change the values of `exp` or of `@I`. The return value of a function in WSL is the final bracketed expression, `(R)`, in the function body.

All that is left to explain is the code that is repeated for each variable in the expression `exp`.

```
IF @N_String(@V(@I)) IN @Domain(store)
THEN @Paste_Over( @ValueOf(store,@N_String(@V(@I))))
FI
```

Each time this point is entered `@I` will be pointing at the abstract syntax tree corresponding to a different variable instance in `exp`. The expression

```
@N_String(@V(@I))
```

selects the name of the current variable from the abstract syntax tree.

The functions `@Domain` and `@ValueOf` used in `@Subst` are not part of `MetaWSL` but are user defined functions in *ConSUS*. `@Domain(s)` returns the set of variables which have been assigned values in the symbolic store, `s`.
The function calls

`@Domain(s)`, and
`@ValueOf(s,name)`

return the value of variable `name` in symbolic store, `s`. `@Paste_Over(x)` is a `MetaWSL` function, which will actually overwrite `@I` with `x`. In this case, this will cause the current variable to be overwritten by its current value in the current symbolic store as required. The return value of a function in WSL is the final bracketed expression, `(R)`, in the function body.

The overall behaviour of the `@Subst` function is now summaried. It takes a store and an arithmetic expression and symbolically evaluates the expression in the store. This is done by replacing the value of each variable in the expression by its current value in the store.

In Figure 5.4, the implementation of *FermaT* `Simplify` as a lightweight theorem prover is shown.

The *FermaT* `Simplify` transformation is applied to the expression `condition`. If the resulting expression is the syntactic object `true` then the boolean value *true* is returned. If *FermaT* `Simplify` fails to transform the condition to `true`, then *false* is returned. The `MetaWSL` function, `@Make`, is used extensively. It constructs abstract syntax trees from components. In this case, the boolean expression `true` has been constructed.

```
MW_BFUNCT @TrueFermat?(condition) ==
VAR < v:=< > > :
@Edit;
@New_Program(condition);
@Trans(TR_Simplify_Item, "");
v := @I;
@Undo_Edit ;
(@Equal?(v,@Make(T_True, < > , < >))) .  ;
```

Figure 5.4: *FermaT* `Simplify` as a light-weight theorem prover

```
MW_BFUNCT @ImpliesFermat?(condition1,condition2) ==
VAR <v:=< > > :
v:=@Make(T_Or,< >, <@Make(T_Not,< >,<condition1>),condition2>);
(@TrueFermat?(v)) .  ;
```

Figure 5.5: Implementation of implication using `MetaWSL`

Implication is equally simple to implement (see Figure 5.5). Given two expressions, `condition1` and `condition2`, to check whether `condition1` implies `condition2`, again the `MetaWSL` `@Make` is used to construct the abstract syntax tree to represent the condition:

```
NOT(condition1) OR condition2
```

and pass it to the `@TrueFermat?` function given in in Figure 5.4.

The *AllImply*(Σ, b) function is used extensively by *ConSUS*. It is implemented as shown in Figure 5.6. It takes two parameters

- `paths`: the list of path conditions, b_i, in conditioned state Σ and
- `conds`: the list of boolean expressions whose ith element is the result, $\sigma_i\ b$, of

```
MW_FUNCT @AllImply(paths, conds) ==
VAR <stop := 1, i:=1 > :
WHILE i <= LENGTH(paths) AND (stop=1)
DO
   IF NOT(@Implies?(paths[i],conds[i]))
   THEN stop:=0
   FI;
   i:=i+1
OD;
(stop).;
```

Figure 5.6: Implementation of *AllImply*

symbolically evaluating b in symbolic store σ_i.

5.3 *ConSUS* with CVC

The Co-operating Validity Checker (CVC) [112] is a successor to the Stanford Validity Checker (SVC) [8], the underlying theorem prover used in *ConSIT* [36]. CVC is a high performance system for checking the validity of formulæ in a relatively rich decidable logic. Atomic formulae are applications of predicate symbols like $<$ and = to first order terms like x and $2 * y + z$. CVC is applicable to boolean expressions made from these atoms. CVC is implemented in about 150K lines of C++. CVC is much more efficient and has more user-friendly input language than its predecessor, SVC.

Queries can be input to CVC in the following form:

```
a,b,c,x : REAL;
QUERY x=x;
QUERY x=(x+1);
QUERY a+x=(x+a);
```

CVC will reply with:

```
Valid.
Invalid.
Valid.
```

An example of an CVC query automatically generated by *ConSUS* when processing the tax example given in Figure 5.2 is:

```
x:REAL;QUERY ((((TRUE) AND ((x) < (50))) AND (TRUE)))
=>
((NOT (((x) >= (75))))) ;
```

In order for *ConSUS* to communicate with CVC, the abstract syntax trees representing WSL conditions have to be converted into inputs like those above. This is very straightforward using the functions of `MetaWSL`. Since every variable in a query has to be declared in CVC, every variable in the WSL condition being processed must be appended to the output string. Apart from this, converting a WSL condition to a CVC query is, in effect, equivalent to writing a pretty printer for WSL conditions. A task which is trivial to implement using `MetaWSL`.

Chapter 6

Conclusion

6.1 Introduction

6.2 Conclusions about research questions or hypotheses

6.3 Conclusions about the research problem

6.4 Implications for theory

6.5 Implications for practice

6.6 Limitations

6.7 Implications for further research

Bibliography

[1] H. Agrawal. On slicing programs with jump statements. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 302–312, Orlando, Florida, June 20–24 1994. Proceedings in SIGPLAN Notices, 29(6), June 1994.

[2] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In 4^{th} *ACM Symposium on Testing, Analysis, and Verification (TAV4)*, pages 60–73, 1991. Appears as Purdue University Technical Report SERC-TR-93-P.

[3] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 23(6):589–616, June 1993.

[4] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, New York, June 1990.

[5] V. Ambriola et al. Symbolic semantics and program reduction. *IEEE Transactions on Software Engineering*, SE-11(8):784–794, August 1985.

[6] T. Ball and S. Horwitz. Slicing programs with arbitrary control–flow. In P. Fritzson, editor, 1^{st} *Conference on Automated Algorithmic Debugging*, pages 206–222, Linköping, Sweden, 1993. Springer. Also available as University of Wisconsin–Madison, technical report (in extended form), TR-1128, December, 1992.

[7] G. N. Balzer, R. and D. Wile. Operational specification as the basis for rapid prototyping. In *Proc. Second Software Engineering Symposium: Workshop on Rapid Prototyping*. ACM SIGSOFT, April 1982.

[8] C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In M. Srivas and A. Camilleri, editors, *Formal Methods In Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, November 1996. Palo Alto, California, November 6–8.

[9] J. Beck and D. Eichmann. Program and interface slicing for reverse engineering. In *IEEE/ACM* 15^{th} *Conference on Software Engineering (ICSE'93)*, pages 509–518. IEEE Computer Society Press, Los Alamitos, California, USA, 1993.

[10] V. Berzins. Software merge: Models and methods for combining changes to programs. *International Journal on Systems Integration*, 1:121–141, August 1991.

[11] J. M. Bieman and L. M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, Aug. 1994.

[12] D. W. Binkley. The application of program slicing to regression testing. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 583–594. Elsevier, 1998.

[13] D. W. Binkley, M. Harman, L. R. Raszewski, and C. Smith. An empirical study of amorphous slicing as a program comprehension support tool. In 8^{th} *IEEE International Workshop on Program Comprehension (IWPC 2000)*, pages 161–170, Limerick, Ireland, June 2000. IEEE Computer Society Press, Los Alamitos, California, USA.

[14] D. W. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology*, 4(1):3–35, 1995.

[15] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT—A formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10(6):234–245, 1975.

[16] R. M. Burstall. Program proving as hand simulation with a little induction. In *Information Processing '74*, pages 308–312. 1974.

[17] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 595–607. Elsevier Science B. V., 1998.

[18] G. Canfora, A. Cimitile, A. De Lucia, and G. A. D. Lucca. Software salvaging based on conditions. In *International Conference on Software Maintenance (ICSM'96)*, pages 424–433, Victoria, Canada, Sept. 1994. IEEE Computer Society Press, Los Alamitos, California, USA.

[19] G. Canfora, A. Cimitile, A. D. Lucia, and G. A. D. Lucca. Decomposing legacy programs: A first step towards migrating to client–server platforms. In 6^{th} *IEEE International Workshop on Program Comprehension*, pages 136–144, Ischia, Italy, June 1998. IEEE Computer Society Press, Los Alamitos, California, USA.

[20] G. Canfora, A. D. Lucia, and M. Munro. An integrated environment for reuse reengineering C code. *Journal of Systems and Software*, 42:153–164, 1998.

[21] J. Cheng. Slicing concurrent programs – a graph–theoretical approach. In P. Fritzson, editor, 1^{st} *Automatic Algorithmic Debugging Conference (AADEGUB'93)*, pages 223–240, 1993. Appears as Springer Lecture Notes in Computer Science vol 749.

[22] J. Choi and J. Ferrante. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems*, 16(4):1097–1113, July 1994.

[23] A. Cimitile, A. De Lucia, and M. Munro. Identifying reusable functions using specification driven program slicing: a case study. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'95)*, pages 124–133, Nice, France, 1995. IEEE Computer Society Press, Los Alamitos, California, USA.

[24] A. Cimitile, A. De Lucia, and M. Munro. A specification driven slicing process for identifying reusable functions. *Software maintenance: Research and Practice*, 8:145–178, 1996.

[25] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.

[26] L. A. Clarke, J. Hassell, and D. J. Richardson. A close look at domain testing. *IEEE Transactions on Software Engineering*, 8:380–390, 1982.

[27] L. A. Clarke and D. J. Richardson. Symbolic evaluation – an aid to testing and verification. In *Proc. of a symposium on Software validation: inspection-testing-verification-alternatives*, pages 141–166. Elsevier North-Holland, Inc., 1984.

[28] A. Coen-Porisini and F. De Paoli. SYMBAD: A symbolic executor of sequential Ada programs. In *IFAC SAFECOMP'90*, pages 105–111, London, 1990.

[29] A. Coen-Porisini, F. De Paoli, C. Ghezzi, and D. Mandrioli. Software specialization via symbolic execution. *IEEE Transactions on Software Engineering*, 17(9):884–899, Sept. 1991.

[30] D. Cohen. Symbolic execution of the gist specification language. In *Proc. of the 8th IJCAI*, pages 16–20, Karlsruhe, Germany, 1983.

[31] D. Cohen. A forward inference engine to aid in understanding specifications. In *Proc. of the National Conference on Artificial Intelegence*, August 1984.

[32] D. Cohen, W. Swartout, and R. Balzer. Using symbolic execution to characterize behavior. *SIGSOFT Softw. Eng. Notes*, 7(5):25–32, 1982.

[33] P. D. Coward. Symbolic execution systems - a review. *Software Engineering Journal*, 3(6):229–239, Nov. 1988.

[34] P. D. Coward. Symbolic execution and testing. *Information and Software Technology*, 33(1):53–64, 1991.

[35] S. Danicic, C. Fox, M. Harman, and R. M. Hierons. ConSIT: A conditioned program slicer. In *IEEE International Conference on Software Maintenance (ICSM'00)*, pages 216–226, San Jose, California, USA, Oct. 2000. IEEE Computer Society Press, Los Alamitos, California, USA.

[36] S. Danicic and M. Harman. Espresso: A slicer generator. In *ACM Symposium on Applied Computing, (SAC'00)*, pages 831–839, Como, Italy, Mar. 2000.

[37] S. Danicic, M. Harman, and Y. Sivagurunathan. A parallel algorithm for static program slicing. *Information Processing Letters*, 56(6):307–313, Dec. 1995.

[38] R. B. Dannenberg and G. W. Ernst. Formal program verification using symbolic execution. *IEEE Trans. Software Eng.*, 8(1):43–52, 1982.

[39] A. De Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviours through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension*, pages 9–18, Berlin, Germany, Mar. 1996. IEEE Computer Society Press, Los Alamitos, California, USA.

[40] Y. Deng, S. Kothari, and Y. Namara. Program slice browser. In 9^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'01)*, pages 50–59, Toronto, Canada, May 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[41] E. W. Dijkstra. *A discipline of programming*. Prentice Hall, 1972.

[42] E. W. Dijkstra. *The characterization of semantics*, chapter 3. Prentice-Hall, 1976.

[43] P. T. F. Ricca. Web application slicing. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM '01)*. IEEE Computer Society, 2001.

[44] S. B. P. Facon. Partial evaluation for the understanding of fortran programs. *International Journal of Software Engineering and Knowledge Engineering*, 4(4):535–559, 1994.

[45] J. Field, G. Ramalingam, and F. Tip. Parametric program slicing. In 22^{nd} *ACM Symposium on Principles of Programming Languages*, pages 379–392, San Francisco, CA, 1995.

[46] J. Field and F. Tip. Dynamic dependence in term rewriting systems and its application to program slicing. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 609–636. Elsevier Science B. V., 1998.

[47] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, Providence, RI, 1967.

[48] C. Fox, M. Harman, R. M. Hierons, and S. Danicic. Backward conditioning: a new program specialisation technique and its application to program comprehension. In 9^{th} *IEEE*

International Workshop on Program Comprenhesion (IWPC'01), pages 89–97, Toronto, Canada, May 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[49] K. B. Gallagher. Evaluating the surgeon's assistant: Results of a pilot study. In *Proceedings of the International Conference on Software Maintenance*, pages 236–244. IEEE Computer Society Press, Los Alamitos, California, USA, Nov. 1992.

[50] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, Aug. 1991.

[51] M. R. Girgis. An experimental evaluation of a symbolic execution system. *Software Engineering Journal*, 7(4):285–290, 1992.

[52] R. Gupta, M. J. Harrold, and M. L. Soffa. An approach to regression testing using slicing. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 299–308, Orlando, Florida, USA, 1992. IEEE Computer Society Press, Los Alamitos, California, USA.

[53] R. Hall. Automatic extraction of executable program subsets by simultaneous program slicing. *Journal of Automated Software Engineering*, 2(1):33–53, 1995.

[54] S. L. Hantler and J. C. King. An introduction to proving the correctness of programs. *ACM Comput. Surv.*, 8(3):331–353, 1976.

[55] M. Harman and S. Danicic. Using program slicing to simplify testing. *Software Testing, Verification and Reliability*, 5(3):143–162, Sept. 1995.

[56] M. Harman and S. Danicic. Amorphous program slicing. In 5^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'97)*, pages 70–79, Dearborn, Michigan, USA, May 1997. IEEE Computer Society Press, Los Alamitos, California, USA.

[57] M. Harman and S. Danicic. A new algorithm for slicing unstructured programs. *Journal of Software Maintenance and Evolution*, 10(6):415–441, 1998.

[58] M. Harman, S. Danicic, and Y. Sivagurunathan. Program comprehension assisted by slicing and transformation. In M. Munro, editor, 1^{st} *UK workshop on program comprehension*, Durham University, UK, July 1995.

[59] M. Harman and K. B. Gallagher. Program slicing. *Information and Software Technology*, 40(11 and 12):577–581, Nov. 1998.

[60] R. M. Hierons and M. Harman. Program analysis and test hyptheses complement. In *IEEE ICSE International Workshop on Automated Program Analysis, Testing and Verification*, pages 32–39, Limerick, Ireland, June 2000.

[61] R. M. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4):233–262, 1999.

[62] R. M. Hierons, M. Harman, C. Fox, L. Ouarbya, and M. Daoudi. Conditioned slicing supports partition testing. *Software Testing, Verification and Reliability*, 12:23–28, Mar. 2002.

[63] S. Horwitz, J. Prins, and T. Reps. Integrating non–interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[64] S. Horwitz, T. Reps, and D. W. Binkley. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 25–46, Atlanta, Georgia, June 1988. Proceedings in *SIGPLAN Notices*, 23(7), pp.35–46, 1988.

[65] W. E. Howden. Dissect-a symbolic evaluation and program testing system. *IEEE Trans. Software Eng.*, 4(1):70–73, 1978.

[66] W. E. Howden. An evaluation of the effectiveness of symbolic testing. *Softw., Pract. Exper.*, 8(4):381–397, 1978.

[67] D. Ince. The automatic generation of test data. *Computer Journal*, 30(1):63–69, January 1987.

[68] D. Jackson and E. J. Rollins. Chopping: A generalisation of slicing. Technical Report CMU-CS-94-169, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1994.

[69] D. B. . A. E. . N. Jones. Partial evaluation and mixed computation. 1987.

[70] T. E. C. Jr., G. H. Holloway, and J. A. Townley. Symbolic evaluation and the analysis of programs. *IEEE Trans. Software Eng.*, 5(4):402–417, 1979.

[71] M. Kamkar. Application of program slicing in algorithm debugging. *Information and Software Technology*, 40(11):637–646, 1998.

[72] R. A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Trans. Software Eng.*, 11(1):32–43, 1985.

[73] R. A. Kemmerer and S. T. Eckman. Unisex: A unix-based symbolic executor for pascal. *Softw., Pract. Exper.*, 15(5):439–458, 1985.

[74] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[75] J. C. King. Program reduction using symbolic execution. *SIGSOFT Softw. Eng. Notes*, 6(1):9–14, 1981.

[76] B. Korel. Identifying faulty modifications in software maintenance. In *Proceedings of 1st International Workshop on Automated and Algorithmic Debugging*, pages 341–356, Linkoping, Sweden, 1993. Springer.

[77] B. Korel. Computation of dynamic slices for unstructured programs. *IEEE Transactions on Software Engineering*, 23(1):17–34, 1997.

[78] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, Oct. 1988.

[79] J. N. . A. E. . W. Kozaczynski. Recovering reusable components from legacy systems by program segmentation. In *Proceedings of 1st Working Conference on Reverse Engineering*, pages 64–72, Baltimore,Maryland,U.S.A, 1993. IEEE Computer Society Press, Los Alamitos, California, USA.

[80] J. Krinke. Static slicing of threaded programs. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pages 35–42, Montreal, Canada, June 1998.

[81] J. Krinke and G. Snelting. Validation of measurement software as an application of slicing and constraint solving. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 661–675. Elsevier, 1998.

[82] A. Lakhotia and J.-C. Deprez. Restructuring programs by tucking statements into functions. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 677–689. Elsevier, 1998.

[83] L. D. Larsen and M. J. Harrold. Slicing object-oriented software. In *Proceedings of the 18th International Conference on Software Engineering*, pages 495–505, Berlin, 1996.

[84] B. K. . J. Laski. Dynamic slicing of computer programs. *The Journal of Systems and Software*, 13(3):187–195, 1990.

[85] P. E. Livadas and A. Rosenstein. Slicing in the presence of pointer variables. Technical Report SERC-TR-74-F, Computer Science and Information Services Department, University of Florida, Gainesville, FL, June 1994.

[86] J. R. Lyle and D. W. Binkley. Program slicing in the presence of pointers. In *Foundations of Software Engineering*, pages 255–260, Orlando, FL, USA, Nov. 1993.

[87] J. R. Lyle and M. Weiser. Automatic program bug location by program slicing. In 2^{nd} *International Conference on Computers and Applications*, pages 877–882, Peking, 1987. IEEE Computer Society Press, Los Alamitos, California, USA.

[88] R. B. . J. Moore. A computational logic. New York, 1979. Academic Press.

[89] M.Ward. Assembler to c migration using the fermat transformation system. *Internation Conference on Software Maintainance*, August 1999.

[90] H. . M.Ward. A multiple backtracking algorithm. *Journal of Symbolic Computation*, (1):1–40, 1994.

[91] L. M. Ott. Using slice profiles and metrics during software maintenance. In *Proceedings of the* 10^{th} *Annual Software Reliability Symposium*, pages 16–23, 1992.

[92] L. M. Ott and J. M. Bieman. Program slices as an abstraction for cohesion measurement. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 681–699. Elsevier, 1998.

[93] L. M. Ott and J. J. Thuss. The relationship between slices and module cohesion. In *Proceedings of the* 11^{th} *ACM conference on Software Engineering*, pages 198–204, May 1989.

[94] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in software development environments. *SIGPLAN Notices*, 19(5):177–184, 1984.

[95] L. C. Paulson. Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*, 828:xvii + 321, 1994.

[96] L. C. Paulson. Isabelle's reference manual. Technical Report 283, University of Cambridge, Computer Laboratory, 1997.

[97] L. C. Paulson. Strategic principles in the design of Isabelle. In *CADE-15 Workshop on Strategies in Automated Deduction*, pages 11–17, Lindau, Germany, 1998.

[98] E. Ploedereder. Symbolic evaluation as a basis for integrated validation. In *Proc. of a symposium on Software validation: inspection-testing-verification-alternatives*, pages 167–185. Elsevier North-Holland, Inc., 1984.

[99] T. Reps. Program analysis via graph reachability. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 701–726. Elsevier Science B. V., 1998.

[100] B. K. . J. Rilling. Case and dynamic program slicing in software maintenance. *International Journal of computer Science and Information Management*, 1998.

[101] B. K. . J. Rilling. Dynamic program slicing methods. volume 40, pages 647–659. Information and Software Technology, 1998.

[102] J. J. . X. Z. . D. Robson. Program slicing for c - the problems in implementation. In *Proceedings of conference on Software Maintenance*, pages 182–190, Sorrento, Italy, 1991. IEEE Computer Society Press, Los Alamitos, California, USA.

[103] P. Rudnicki. What should be proved and tested symbolically in formal specifications? In I. C. Society, editor, *4th Intern. Workshop on Software Specification and Design*, pages 190–195, California, 1987.

[104] N. W. . M. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[105] M. K. . P. F. . N. Shahmerhi. Interprocedural dynamic slicing applied to interprocedural data flow testing. In *Proceedings of conference on Software Maintenance*, pages 386–395, Montreal Quebec, Canada, 1993. IEEE Computer Society Press, Los Alamitos, California, USA.

[106] M. K. . P. F. . N. Shahmerhi. Three approaches to interprocedural dynamic slicing. *Journal of Microprocessing and Microprogramming*, 38:625–636, 1993.

[107] S. Sinha, M. J. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control-flow. In *Proceedings of the 21st International Conference on Software Engineering*, pages 432–441. ACM Press, May 1999.

[108] D. M. Solis. The unisex system - a symbolic executor for the pascal language. Master's thesis, Dept. of Computer Science, Univ. of California, Santa Barbara, California, 1982.

[109] J. E. Stoy. *Denotational semantics: The Scott–Strachey approach to programming language theory*. MIT Press, 1985. Third edition.

[110] N. W. . J. G. . T. G. . D. Strasburg. Locating user functionality in old code. In *Proceedings of Conference on Software Maintenance*, pages 200–205, Orlando,FL,USA, 1992. IEEE Computer Society Press, Los Alamitos, California, USA.

[111] A. Stump, C. W. Barrett, and D. L. Dill. CVC: a cooperating validity checker. In J. C. Godskesen, editor, *Proceedings of the International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science, 2002.

[112] W. R. Swartout. Gist english generator. In *AAAI*, pages 404–409, 1982.

[113] W. R. Swartout. The gist behavior explainer. In *AAAI*, pages 402–407, 1983.

[114] F. Tip, J.-D. Choi, J. Field, and G. Ramalingham. Slicing class hierarchies in C++. In *Proceedings of the* 11^{th} *conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'96)*, San Jose, Oct. 1996.

[115] G. A. Venkatesh. The semantic approach to program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–28, Toronto, Canada, June 1991. Proceedings in *SIGPLAN Notices*, 26(6), pp.107–119, 1991.

[116] M. K. W. Farmer, in W. Farmer and M. Kohlhase. Mechanizing the traditional approach to partial functions. In *Proceedings of the Workshop on the Mechanization of Partial Function*, 1996.

[117] M. Ward. *Proving Program Refinements and Transformations*. DPhil Thesis, Oxford University, 1989.

[118] M. Ward. Reverse engineering through formal transformation. *The Computer Journal*, 37(5), 1994.

[119] M. Ward. Derivation of data intensive algorithms by formal transformation: The Schnorr–Waite graph marking algorithm. *IEEE Transactions on Software Engineering*, 22(9):665–686, Sept. 1996.

[120] M. Ward. Assembler to C migration using the FermaT transformation system. In *IEEE International Conference on Software Maintenance (ICSM'99)*, Oxford, UK, Aug. 1999. IEEE Computer Society Press, Los Alamitos, California, USA.

[121] M. Ward. The formal approach to source code analysis and manipulation. In 1^{st} *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 185–193, Florence, Italy, 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[122] M. Ward, F. W. Calliss, and M. Munro. The maintainer's assistant. In *Proceedings of the International Conference on Software Maintenance 1989*, page 307. IEEE Computer Society Press, Los Alamitos, California, USA, 1989.

[123] M. Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[124] M. Weiser. Programmers use slicing when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[125] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[126] M. Weiser and J. R. Lyle. *Experiments on slicing–based debugging aids*, chapter 12, pages 187–197. Empirical studies of programmers, Soloway and Iyengar (eds.). Molex, 1985.

[127] L. J. White and E. I. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, 6:247–257, 1980.

[128] N. Wirth. An assessment of the programming language pascal. *IEEE Trans. Software Eng.*, 1(2):192–198, 1975.