ConSUS: A Light-Weight Program Conditioner

Sebastian Danicic, [a] Mohammed Daoudi, [a] Chris Fox, [c]
Mark Harman, [b] Rob M. Hierons, [b] John R. Howroyd, [a]
Lahcen Ourabya, [a] and Martin Ward [d]

[a] *Department of Computing, Goldsmiths College, University of London, New Cross, London SE14 6NW, United Kingdom*

[b] *Department of Information Systems and Computing, Brunel University, Uxbridge, Middlesex, UB8 3PH, United Kingdom.*

[c] *Department of Computer Science,University of Essex, Wivenhoe Park, Colchester,CO4 3SQ, United Kingdom*

[d] *Software Technology Research Lab, De Montfort University, The Gateway, Leicester LE1 9BH, United Kingdom*

Abstract

Program conditioning consists of identifying and removing 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, re–use and re–engineering. Program conditioning relies upon both symbolic execution and reasoning about symbolic predicates. Automation of the process therefore requires some form of automated theorem proving. However, the use of a full-power 'heavyweight' theorem prover would impose unrealistic performance constraints.

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

The paper also provides empirical evidence that conditioning produces a significant reduction in program size and, although exponential in the worst case, the conditioning system has low degree polynomial behaviour in many cases, thereby making it scalable to unit level applications of program conditioning.

Key words: program conditioning, slicing, program transformation, decision procedures

1 Introduction

1.1 Related Work

Program slicing originated with Mark Weiser's 1979 doctoral thesis [1]. Weiser's formulation of slicing [1,2] was a static one, captured by the slicing criterion, which was a set of variables and a program point. This static paradigm for slicing, was augmented by a dynamic paradigm in 1988, by Korel and Laski [3]. Dynamic slicing represented the first move away from the static paradigm, indicating that there may be other ways to formulate a slicing criterion.

The move from static to other paradigms was further developed in 1990, by Venkatesh, who suggested a form of slicing to provide a bridge between static and dynamic slicing, called quasi-static slicing. A quasi-static slice is constructed with respect to a prefix of input, rather than a full input sequence. Quasi-static slicing was motivated by work on partial evaluation [4,5]. The concept of quasi-static slicing is subsumed by conditioned slicing, which is the most general form of slicing, hitherto explored in the slicing literature.

Conditioned slicing, the subject of this paper, was introduced by Canfora et al. [6] in 1994. Field, Ramalingam and Tip [7] introduced a similar technique called constrained slicing, which also uses conditions to specialize programs. In constrained slicing, parts of the program which cannot contribute to the values of variables of interest are identified as holes. Because it contains holes, a constrained slice is not necessarily an executable subprogram, as a conditioned slice is.

Earlier, ideas and techniques very similar to program conditioning were introduced by Coen-Porisini et al. in [14]. This work uses symbolic execution and theorem proving to specialize Ada programs. In their approach, generalised software components are specialised by restricting their domain of inputs thereby increasing the efficiency of components for each particular instance of their use.

To implement conditioned slicing it is necessary to use some form of theorem proving technology. In early work on conditioned slicing, the theorem proving was performed by hand [6,8,9], or by term-rewriting (in the case of constrained slicing [7]). More recent work explored the use of Isabelle [10] and SVC [11]. This work using Isabelle and SVC can be thought of as a proof-of-concept implementation, showing how the theorem prover could be interchanged as a component of the overall conditioning approach. The present work explores the use of the FermaT `simplify` transformation, a decision procedure which is used to implement a form of highly light-weight theorem proving.

The paper also introduces a new approach to symbolic execution which follows a different form to previous symbolic executors [11–14] in its handling of loops and

which is more closely integrated with the theorem prover than previous approaches [9–11] [1] , thereby exploiting the possibility of path–pruning to speed up the conditioning process. Our approach is similar to .

Other authors have considered the way in which theorem provers may be used in slicing [15] and the way in which forms of symbolic execution can assist related analyses such as constrained test data generation [16–18].

In the VALSoft project [15], Krinke and Snelting show how the computation of symbolic values can be used to refine data-flow relations between array assignments and uses. Specifically, they consider the situation where an assignment to an array like

```
A[i] := 4;
```

may not filter through to a reference to the same array

```
IF A[j] == K THEN ...
```

because it can be statically determined that `i` and `j` are guaranteed to be non-equal at the `IF` statement. In order to determine this statically, Krinke and Snelting use a simple form of symbolic execution and then a theorem prover to determine (statically) the properties of expression indices at array expressions. This work differs from the work reported here because the application of theorem prover and symbolic executor is used to refine the *static* data dependence relation used to construct a *static* slice.

In the work of Offutt et al. [16–18], constraint solving techniques are used to support automated test data generation. This approach requires the backward propagation of constraints from a point of interest, p, within the program. This is a form of backward symbolic execution. The constraints are (ideally) propagated back to the starting state of the program, where they become constraints on the input space. The solution of these input constraints is a valid test data vector to execute point p in the desired way. Constraint solving can be achieved using a theorem prover or a simple decision procedure, offering similar speed/precision trade–offs to those considered in the present paper.

There are several surveys of slicing in the literature which cover dynamic slicing and its applications [19], slicing techniques and algorithms [20], forms of slicing and their applications [21–23] and empirical results on the application of slicing [24].

[1] It has some similarities to the approach taken by Coen-Porisini et al. in [14]. See Section 3.

1.2 Program Conditioning

Program conditioning [2] [9,25], like program slicing [2], is a form of source code manipulation that allows a software engineer to extract an executable sub-program based upon a criterion of interest. The original formulation of slicing [2] was static. That is, the slicing criterion contained no information about the input to the program. A static end-slice of program P with respect a set of variables V, is a program P' that 'behaves the same' as P with respect to all the variables in V. Furthermore, P' is obtained from P by statement deletion. The way in which slicing produces an executable sub-program, based upon some criterion of interest, gives rise to many applications. For example, slicing has been applied to, among others, debugging [26,27], testing [28–31], program comprehension [32–34], program decomposition [35] and integration [36,37], software metrics [38–40] and re-engineering and reverse engineering [6,41–43].

Static slicing has now reached a mature stage of development, in which tools such as the Wisconsin Program Slicing System, marketed through Grammatech [44] can efficiently slice real-world C programs of the order of hundreds of thousands of lines of code in reasonable time [45]. Conditioning is central in a variation 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 captures a set of initial program states of interest. 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 is executed in an initial state which satisfies the condition.

Conditioning is defined independently of slicing: The result of conditioning P will be a program, Q, that behaves the same as P whenever the inputs satisfy a path condition. A path condition is simply a boolean expression involving some or all of the program variables. In the approach described in this paper conditions of interest are expressed as Assert statements, i.e. boolean expressions, which may be arbitrarily placed throughout the program. In this case, the resulting program, Q, must agree with original, P, for all inputs where P satisfies these intermediate Assert statements. Even where no Assert statements are added, the system will attempt to remove infeasible paths (a useful step in itself).

The paper focuses upon the conditioning step in producing a conditioned slice. This is because the slicing step is standard static slicing, for which a WSL static slicer

[2] Henceforth, the phrase 'program conditioning' will be referred to simply as 'conditioning'.

is used, which implements an extension [46] of Hausler's [47] data-flow based approach to static slicing. The static slicing phase is not discussed further. It is the conditioning phase that makes conditioned slices interesting, smaller than their static counter-parts (in general) and, crucially, which increases the difficulty involved in slice construction (because both symbolic execution and some form of theorem proving is required).

As an example of the way in which conditioning identifies sub-programs, consider the Taxation program in Figure 1. The figure contains a fragment [3] of a program which encodes the UK tax regulations in the tax year April 1998 to April 1999. Each person has a personal allowance which is an amount of un-taxed income. 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 sub-program extracted 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.

In the worst case, there is no doubt that the time to perform 'best possible' program conditioning will be exponential in the size of program being conditioned. This fact is inherent to the problem of conditioning. It can be seen that by considering a sequence of simple `IF-THEN-ELSE` statements. The number of paths is clearly 2^n. If no pruning is possible, then the number of calls to the theorem prover will therefore be 2^n. The same, however, is also true of many aspects of theorem proving. For example, the Boolean Satisfiability Problem, which the CHAFF [48] solver at the heart of the CVC theorem prover [49] implements a solution to, is a well known NP-Complete problem [50]. As in the case of theorem provers, this worst case scenario does not imply that implementations of conditioners are infeasible. The reasons for this are twofold:

(1) In conditioning, *any* resulting reduction in program size represents progress. Even if the best possible results require exponential time, it is possible that significant reductions will be performed more quickly.
(2) In many cases conditioning may be performed in low order polynomial or even quadratic time as suggested by the empirical study in this paper. In such cases conditioning may be applicable to unit level applications at least.

The language of implementation and the language which is sliced by the approach

[3] This is WSL version of the C program previously used in [10].

```
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
```

Conditioned program: boxed lines of code

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

Fig. 1. A Fragment of the Taxation Calculation Program in WSL

`SKIP`	(do nothing)
{ B }	(Assert statement)
`ABORT`	(abnormal termination)
x := e	(assignment)
S_1; S_2	(sequence)
`IF` B `THEN` S `FI`	(`IF-THEN`)
`IF` B `THEN` S_1 `ELSE` S_2 `FI`	(`IF-THEN-ELSE`)
`IF` B_1 `THEN` S_1 `ELSIF` $\cdots$ `ELSIF` B_n `THEN` S_n `FI`	(guarded command)
`WHILE` B `DO` S `OD`	(repetition)

Fig. 2. WSL Syntax Used in this Paper

reported here is WSL, the Wide Spectrum Language, introduced by Ward [51] for reverse engineering [52] through program transformation [53,54] (See Section 4.1 for a discussion on why WSL was chosen for this project.). WSL uses an Algol-like syntax, but has additional facilities to make it wide-spectrum and to allow transformations to be expressed within WSL itself. The subset of WSL syntax used in this paper is described in Figure 2. WSL also has `IF-THEN` and `IF-THEN-ELSE` statements. These are, however, just syntactic sugar for special cases of the guarded command where the number of guards is one and two respectively. The Assert statement B is equivalent to `IF` B `THEN SKIP ELSE ABORT`. It is these Assert statements that are used to express the conditions of interest used in conditioning.

The contributions of this paper are:

- To define a new more efficient algorithm for program conditioning.
- To report on an empirical studies which demonstrate that:

(a) On small 'real programs' *ConSUS* produces a considerable reduction in program size when used with and without a program slicer.
(b) The *ConSUS* algorithm when used in conjunction with WSL's *FermaT* `Simplify` has the potential for 'scaling up' for use on large systems.

The rest of this paper is organised as follows:

Section 2 introduces and defines conditioning. In Section 3, the new conditioning algorithm is defined in detail. Section 4 describes features of the implementation in more detail. In particular, subsection 4.1 briefly introduces WSL, the language used for implementing *ConSUS*. The *FermaT* `Simplify` transformation is described. In Section 4.2 details of the implementation of *ConSUS* and are given. It is shown

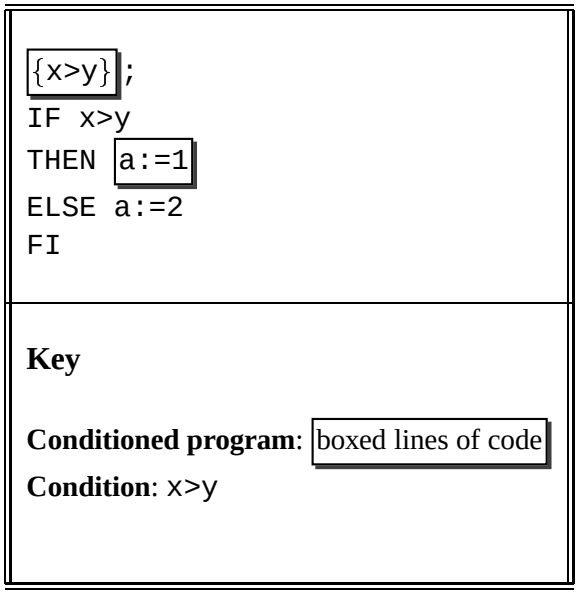


Fig. 3. Conditioning a Simple Program

how the novel features of WSL facilitate many of the programming tasks. In Section 5, our empirical studies are presented. Conclusions and directions for future work are presented in Section 6.

2 Conditioning

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 are expressed by adding Assert statements to the program being conditioned. Consider, for example the program in Figure 3. Here, program simplification is being attempted assuming that it 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 original program's behaviour and that of the conditioned program are identical.

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 4 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")`.

A 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

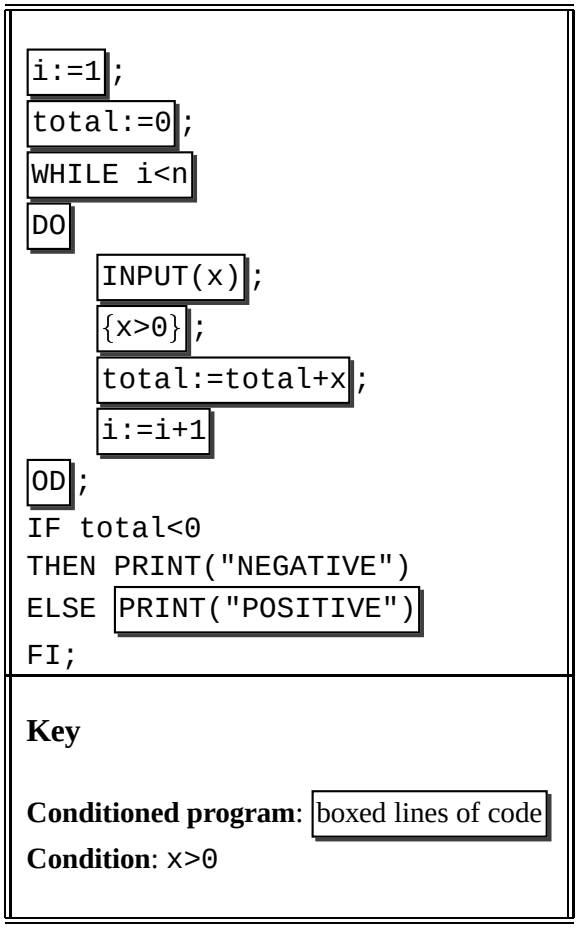


Fig. 4. Conditioning with Intermediate Asserts

the program contains no Assert statements (see Figure 5 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 appears sensible to utilise existing automated theorem provers rather than to develop new ones. Consider the program in Figure 6. Here, conditioning of a simple `IF` statement assuming that the initial state has the property that $x > y \wedge y > z$ 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 $x > y \wedge y > z \implies x > z$. 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 precision of the underlying theorem prover which determines the truth of

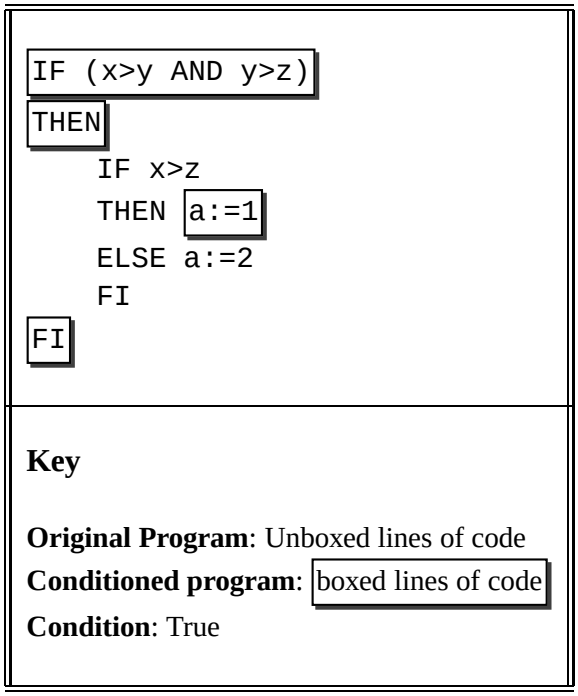


Fig. 5. Conditioning without Assert

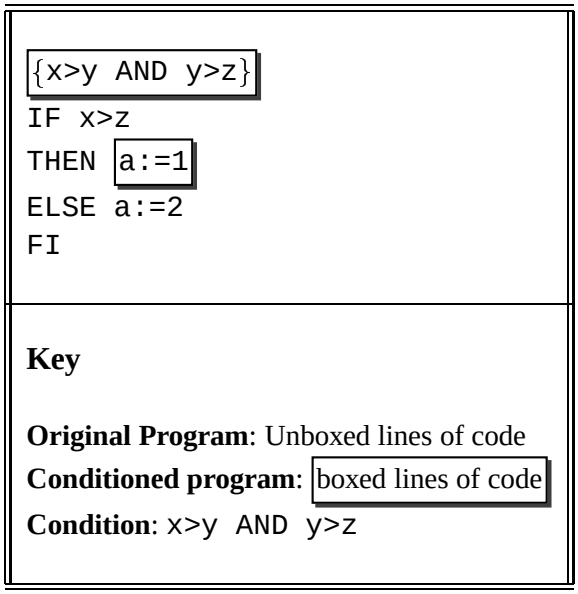


Fig. 6. Conditioning a Simple Program

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 programming 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' inequalities.

3 The *ConSUS* Conditioning Algorithm

3.1 An Overview of the Approach

When implementing an interpreter, a program is evaluated in a state, which maps variables to their values [55]. In symbolic execution [13,56–58], the state, called a *symbolic store* [4] , maps variables, not to *values*, but to *symbolic expressions*. The expressions are the objects that can occur on the right hand sides of assignment statements (in this paper, it is assumed that these are simply arithmetic expressions).

When a program is symbolically evaluated in an initial symbolic store it gives rise to a set of possible final symbolic stores. The reason that a symbolic evaluator returns a *set* 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 set 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 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 is simply a boolean expression involving constants and variables of the program.

A *conditioned state*, Σ, is represented by a set of path condition, symbolic store pairs. A pair (b, σ) being an element of a conditioned state implies that the symbolic store σ can be reached if path condition b is true initially. If a conditioned state contained the pair (false, σ) this is 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 [5] . In

[4] Usually called the *symbolic state*.

[5] In [14], 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.

practice, a conditioner will normally be applied to programs starting in the *natural conditioned state*. In the natural conditioned state, the corresponding symbolic store, *id* 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.

3.1.1 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, therefore, the final path condition, is equivalent to false (a contradiction) this means that the store is not reachable. If, on the other hand , the final path condition is equivalent to true (a tautology) then all paths starting from the given initial store will lead to the corresponding symbolic store. 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, to do this perfectly is not a computable problem and therefore, in general, infeasible paths will be unnecessarily considered.

Consider again, the program in Figure 6. This program potentially has two possible final symbolic stores:

$$[a \rightarrow 1]$$
$$[a \rightarrow 2]$$

The corresponding path conditions are:

$$x > y \quad \wedge \quad y > z \quad \wedge \quad x > z$$

$$x > y \quad \wedge \quad y > z \quad \wedge \quad \neg(x > z).$$

Combining these two gives the conditioned state with two elements:

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

$$(x > y \quad \wedge \quad y > z \quad \wedge \quad \neg(x > z), \quad [a \rightarrow 2]).$$

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

Often programs containing no Assert statements will be conditioned. This corresponds to removing *dead* code. Consider the program in Figure 5. The programs in Figures 6 and 5 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 in 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 clearly an improvement over other systems like *ConSIT* [10] 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 [14]. Their program specializer, 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 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 that 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, wrong information will never be generated, just less precise information.

The approach taken by *ConSUS* (based on the approach of *ConSIT* [10]) is less crude, however. In this case, as a result of symbolically evaluating a `WHILE` loop, there arises 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 these two approaches differ, consider the example given in Figure 7. 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

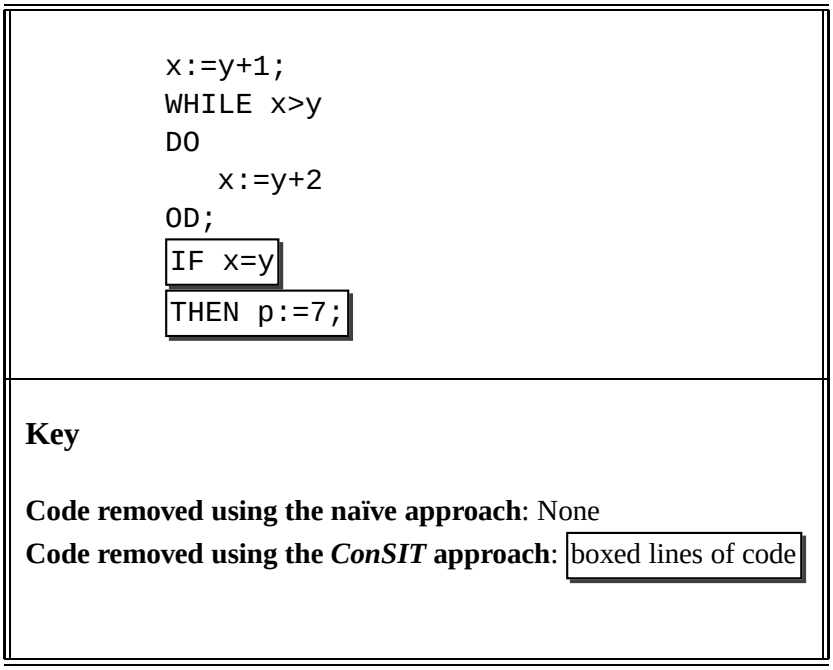


Fig. 7. Conditioning a `WHILE` loop using two Approaches

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.

3.2 The ConSUS Algorithm in Detail

In this section, the algorithm used by *ConSUS* is explained in detail. For each WSL syntactic category given in Figure 2, 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 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*. The method by which *prove* is actually implemented for *FermaT* `Simplify` will be given in Section 4. Given a conditioned state, Σ, and a boolean expression, b, the following expressions will also be useful throughout our description:

- *AllImply*(Σ, b)
  Denoting that 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.
- *AllFalse*(Σ)
  Denoting that for all pairs (c, σ) in conditioned state Σ, *prove*$(\neg c)$ evaluates to *true*. If the conditioner has reached a point whose state, Σ, satisfies *AllFalse*(Σ), then all code following this point is unreachable and can thus be removed.

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.

3.2.1 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 (*false*, *id*).

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

For all statements s, define

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

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

3.2.2 Conditioning SKIP

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

Conditioning a SKIP has no effect.

3.2.3 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	$\mathit{AllImply}(\Sigma, b)$	The assert condition will always be *true*
2	$\mathit{AllImply}(\Sigma, \texttt{NOT}\ b)$	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 $(b_i\ \texttt{AND}\ \sigma_i\ b, \sigma_i)$ where b_i AND $\sigma_i b$ is the boolean expression created by 'ANDing' the boolean expression b_i with the result of symbolically evaluating the boolean expression [6] b in symbolic store σ_i.

This represents the fact a program will continue executing after an Assert statement in stores where b evaluates to *true*. Formally, in this case,

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

[6] 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 $(b_i\ \texttt{AND}\ \sigma_i b)$ is the boolean expression: $a + z = 5\ \texttt{AND}\ z + 1 > 17 + 1$.

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

3.2.4 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. This is very straightforwardly achieved using *FermaT* transformations as will described in Section 4. Given a symbolic store, σ, we use standard notation $\sigma[x \to e]$ 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:

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

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

3.2.5 Conditioning Statements Sequences

In the case of standard semantics [55], 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.

3.2.6 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 [59], 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 conditioning guarded commands, it is convenient to represent a guarded command as

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

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

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

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

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:

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

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

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:

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

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

$$\Sigma_{j+1} = \bigcup_{i=1}^{n} \{(b_i \ \mathsf{AND} \ \sigma_i \ B_j, \sigma_i)\}.$$

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*$(\Sigma_j, \mathsf{NOT} \ B_j)$ 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*$(\Sigma_j, \mathsf{NOT} \ B_j)$ then it cannot said for certain whether B_j will be chosen or not. This is represented by keeping the guarded, *statement*$(\Sigma_j, B \rightarrow S_j)$, 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* [10], where all paths were fully expanded before any simplification took place.

3.2.7 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.*

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

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.*

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

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 3.1.

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 loop in state Σ assuming that the loop terminates.

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

	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

Fig. 8. `WHILE` loop possibilities

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 8.

Each case in Figure 8 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,

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

and

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

will have different values (Figures 9 and 10). 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

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

Fig. 9. `WHILE` loop final states in each case

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

Fig. 10. `WHILE` loop resulting statements in each case

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 non terminates. 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'.

`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*

Fig. 11. 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*

Fig. 12. Conditioning a `WHILE` loop (Case 3)

Case 6: the loop does not terminate. The final state is $\perp$ 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*.

3.3 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 11 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:=2`
`OD`
	`IF x=1`
	`THEN x:=2`
	`FI`

Original Program	Output from *ConSUS*

Fig. 13. 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*

Fig. 14. Conditioning a `WHILE` loop (Case 5)

In Figure 12 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 therefore the following IF statement can be simplified.

The program in Figure 13 has a while loop which is either not executed at all or exactly once. *ConSUS* replaces it with an IF. 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 14, although the loop itself cannot be simplified, *ConSUS* recognises that the loop must be executed at least once and hence the later IF can be simplified.

In Figure 15, *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.

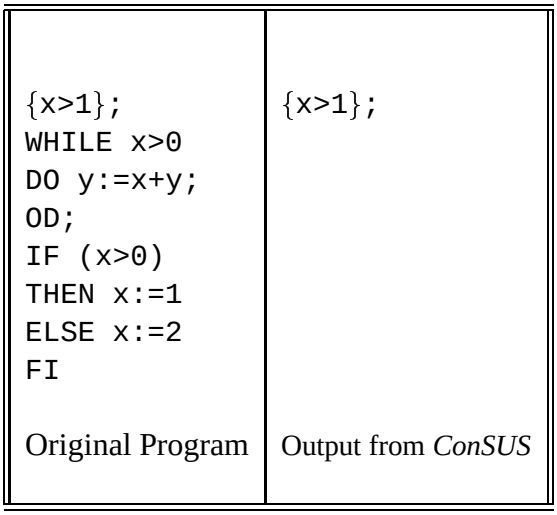


Fig. 15. Conditioning a `WHILE` loop (Case 6)

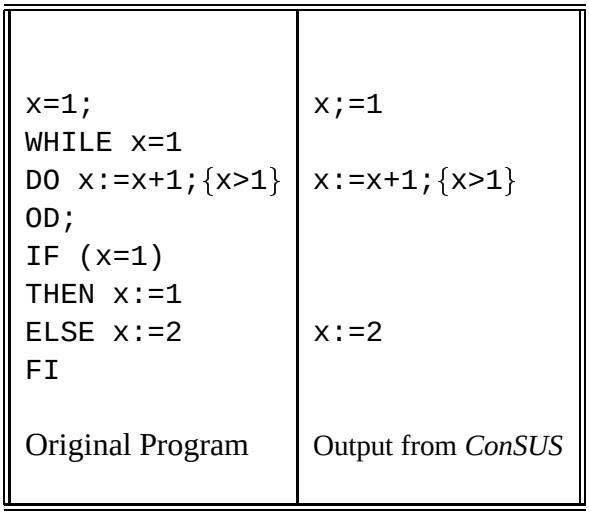


Fig. 16. Conditioning a `WHILE` loop (Case 7)

In Figure 16 we have 'helped' the theorem prover with some 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 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 give us the single statement `x:= 2`.

4 The Implementation

4.1 WSL and FermaT

The *ConSUS* system is implemented using WSL, the Wide Spectrum Language, introduced by Ward [51] for reverse engineering [52–54]. 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 as described earlier.
(2) Within WSL is another language `MetaWSL` [51], 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.

4.1.1 Theoretical Foundations of FermaT

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

The WSL language was developed in parallel with the development of a transformation theory and proof methods. During this time the language has been extended from a simple and tractable kernel language [51] to a complete and powerful programming language. At the 'low-level' end of the language there exists 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.

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 has been tractable directly. More recent work [53] has demonstrated the benefits for representing PLC (programmable logic controller) code and migrating x86 embedded systems to generic C code.

4.1.2 Description of 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 valid (other than assertions and comments). Finally it deletes `SKIP` statements within the current item:

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 an expression and condition simplifier were:

(1) Efficient execution: especially on small expressions. This implies a short start up time;
(2) Easily extensible. 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. 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) 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 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).

4.2 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 state is a list of symbolic store, path condition pairs as described in Section 3. 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 17, 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`. (Variables are not typed in WSL).

```
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 value of the special global variable, `@I` on to the stack and temporarily assigning `@I`, to the syntax tree corresponding to the expression `exp`. The `@Undo_Edit`

```
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 ) . ;
```

Fig. 17. Evaluating an Expression in a Symbolic store

command pops of the previously stacked value into `@I`. This useful technique can be used for all elements of abstract syntax: 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 syntactic 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 17

```
@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))))
```

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 to 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`.

`@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.

5 Empirical Validation

There are two sections to the empirical validation. First, measurements of the effectiveness of *ConSUS* when applied to small but realistic programs are performed. The reduction in program size produced both by conditioning alone and also produced by combining conditioning with slicing are given. It is demonstrated that in both cases conditioning using *ConSUS* produces a considerable reduction in program size.

Secondly, the scalability of *ConSUS* is examined. Programs are constructed using multiple repetitions of one of some fixed program fragments. This enables arbitrarily large programs to be generated using these fragments. *ConSUS* is timed on successive instances of various combinations of these fragments. Graphs of *ConSUS* execution time against program size are produced and analysed. It is shown

that the technique employed by *ConSUS* appears to scale well at least at the unit level.

5.1 Effectiveness

In this section, the behaviour of *ConSUS* when applied to more realistic applications written in WSL is analysed. Examples of the decrease in program size when *ConSUS* is used on its own and also when it is combined with a slicer are given for a variety of slicing criteria. The programs [7] considered are:

(1) Student Marks Processor
(2) A Calendar Program
(3) Tax Allowance Calculator

As with other work on empirical aspects of slicing [60–63], the slicing criteria are developed to be realistic criteria for the programs selected to typical the kinds of query a user of a slicing system might present for the programs under consideration.

5.1.1 Student Marks Processor

The student marks processor is a 230 line WSL program which allows the user to enter marks for three courses for an arbitrary number of students. It outputs a variety of statistics including the final degree classification for each student, the average, highest and lowest marks for each course and the number of students receiving each degree classification. The program has error checking in the sense that it uses loops to force the user to input marks in the correct range (0-100).

The program is 'decorated' with a number of ASSERT statements of the form `{read>=0 AND read<=100}`. These correspond to the assumption that all inputs to the program are correct at the first attempt. When conditioned with respect to these criteria *ConSUS* reduces the original 230 line program to 132 lines. All the unnecessary loops that force input to be in the correct range are correctly removed by *ConSUS*. This corresponds to a reduction in size as a result of conditioning of almost 43%.

In the next set of experiments, the original program is first sliced with respect to a number of different criteria and then each of these slices is further conditioned. This reduction in size produced by conditioning after slicing is then recorded.

For example, the original program, was first sliced at the end with respect to the variable `numfirsts` to give rise to a 31 line program. The resulting slice was

[7] All the examples can be found at http://www.doc.gold.ac.uk/~mas01sd/consus/.

Conditioning Criterion	Slice Variable	Size of Original	Size after slicing	Size after conditioning	Reduction due to *ConSUS*
Inputs correct	No slicing	230 lines	230 lines	123 lines	43%
Inputs correct	`numfirsts`	230 lines	31 lines	21 lines	32%
Inputs correct	`average3`	230 lines	29 lines	19 lines	34%
Inputs correct	`numstudents`	230 lines	5 lines	3 lines	40%
Inputs correct	`averagegrade1`	230 lines	41 lines	32 lines	22%
Inputs correct	`lowest1`	230 lines	39 lines	24 lines	38%

Fig. 18. Size of Student Marks program after first slicing and then conditioning

then conditioned with respect to the same conditioning criterion. Again, all the unnecessary loops that force input to be in the correct range were removed by *ConSUS*. The resulting conditioned slice, thus, has the same effect as the original on the variable `numfirsts` assuming the all inputs are correct first time. In this example there was almost a 1/3 reduction in the size of the slice as a result of conditioning after slicing.

Similarly, slicing with respect to variable `average3` leaves a 29 line program which is further reduced to 19 by conditioning. Figure 18 shows the results of these experiments.

5.1.2 The Calendar Program

The input to this 123 line WSL program is any day since first of January, 1 A.D. The output is the number of days since the first of January 1 A.D. together with the day of the week of the input date. The program takes account of the 11 'lost days' between 2 and 14 September 1752 [64] and the new rules for deciding on leap years that came into effect around 1800.

Again, measurements of the decrease in program size when *ConSUS* is used on its own and also when it is combined with a slicer were made.

The conditioning criteria are:

(1) that all inputs are first time correct (as in the Student Marks Processor Program) and
(2) that all inputs are first time correct and that the year is after the dreaded 1752.

and the slices were taken at the end of the program with respect to variables:

(1) `leap`
(2) `dayofweek`
(3) `month`

Conditioning Criterion	Slice Variable	Size of Original	Size after slicing	Size after conditioning	Reduction due to *ConSUS*
1	`No Slicing`	123 lines	123 lines	100 lines	19%
2	`No Slicing`	123 lines	123 lines	95 lines	23%
1	`leap`	123 lines	36 lines	32 lines	11%
2	`leap`	123 lines	36 lines	32 lines	11%
1	`dayofweek`	123 lines	90 lines	79 lines	12%
2	`dayofweek`	123 lines	90 lines	78 lines	13%
1	`month`	123 lines	21 lines	13 lines	38%
2	`month`	123 lines	21 lines	13 lines	38%

- Conditioning Criterion 1 corresponds to all inputs being initially in the correct range.
- Conditioning Criterion 2 corresponds to all inputs being initially in the correct range and the input year being after 1752.

Fig. 19. Size of Calendar program after first slicing and then conditioning

Without first slicing, using the first conditioning criterion, *ConSUS* successfully removes all the 'force input loops' as before, reducing the program from 123 to 100 lines and using the second conditioning criterion, *ConSUS* also removes the code that specifically checks whether the date falls within the 'lost eleven days', reducing the original from 123 to 95 lines.

Slicing the program with respect to the variables above and then conditioning produces a further reduction in program size as can be seen in Figure 19.

5.1.3 The Tax Allowance Calculator

The Tax Allowance Calculator is a 129 line program which asks the user a number of questions from which it then calculates the income tax allowance, tax code etc. for the user. It is an encoding of the UK Tax rules as they were between April 1998 and April 1999. Tax codes and allowances vary depending on the state of an individual. Important criteria include marital status, age and sex. A blind person gets at a different allowance from a sighted person. This program is first sliced with respect to variables

(1) `code`
(2) `pc10`
(3) `personal`

Each slice was then conditioned with respect to the criteria:

(1) `age > 65`
(2) `blind = 1`

Conditioning Criterion	Slice Variable	Size of Original	Size after slicing	Size after conditioning	Reduction due to *ConSUS*
1	No Slicing	129 lines	129 lines	71 lines	45 %
2	No Slicing	129 lines	129 lines	71 lines	45 %
3	No Slicing	129 lines	129 lines	69 lines	47 %
1	`code`	129 lines	48 lines	24 lines	50 %
2	`code`	129 lines	48 lines	18 lines	63 %
3	`code`	129 lines	48 lines	24 lines	50 %
1	`pc10`	129 lines	42 lines	27 lines	36 %
2	`pc10`	129 lines	42 lines	26 lines	38 %
3	`pc10`	129 lines	42 lines	19 lines	55%
1	`personal`	129 lines	37 lines	27 lines	27%
2	`personal`	129 lines	37 lines	26 lines	30%
3	`personal`	129 lines	37 lines	24 lines	35%

- Conditioning Criterion 1 corresponds to the input age being greater than 65.
- Conditioning Criterion 2 corresponds to the input person being blind.
- Conditioning Criterion 3 corresponds to the input person being not married.

Fig. 20. Size of the Tax program after first slicing and then conditioning

(3) `married = 0`

As in the other examples, figure 20 shows a significant reduction in program size results from both from just conditioning and also conditioning after slicing.

5.1.4 Summary

Conditioning the 'small but realistic' programs in our study using *ConSUS* resulted in an average reduction in program size of approximately 35%. The maximum reduction 63% and the minimum reduction was 11%. The average reduction in non-sliced programs was 37% whereas the average for sliced programs was 34%. The conditioning criteria were not arbitrary but represented 'meaningful' properties of the input state of the programs. These figures show that conditioning can produce a considerable reduction in program size when applied to realistic programs. Clearly more work is required to scale the application of conditioning to larger programs. however, it should be stressed that conditioning is inherently harder than traditional static slicing due to the requirement of symbolic execution and theorem proving and so the authors believe that these initial results on small programs are encouraging, going some way to demonstrating the 'proof of concept' for conditioned slicing.

Six classes of programs called F, T, SN, NSN, SSC, and NSSC are considered. The programs in each class are formed from 'generative' program fragments with multiple repetitions of one of these fragments. The classes F, T, SN, NSN are generated from the 'base' programs given in Figure 21. The SSC and NSSC are generated from fragment given in Figure 22. This gives us a systematic approach to testing the scalability of *ConSUS*.

The programs of class F are generated from the fragments shown in Figure 21 with multiple repetitions of the second fragment.

```
y:=1; x:=0;

IF x=0
THEN IF y=2
     THEN n:=3
     ELSE n:=5
     FI
ELSE n:=5
FI;
IF n=3
THEN a:=4
ELSE a:=5
FI;
```

F-Style Program

```
y:=0; x:=1;
c:=2; b:=c+1;

IF b > c
THEN x:=x+1; y:=y+2;
     b:=b+1; c:=c+2
ELSE x:=x+2; y:=y+1;
     b:=b+2; c:=c+1
FI;
IF (b > c) OR (x > y)
THEN x:=x+2; y:=y+1;
     b:=b+2; c:=c+1
ELSE x:=x+1; y:=y+2;
     b:=b+1; c:=c+2
FI ;
```

T-Style Program

```
x:=0;
b:=1;
c:=2;

IF x <= b
THEN
   x:=x+1;
   b:=b+1;
   c:=c+1;
   IF b<=c
   THEN
      y:=1
   FI
ELSE y:=2
FI;
```

SN-Style program

Fig. 21. The three considered classes of program

This set of programs tests the conditioning process of *ConSUS* on sequential `IF` statements where the predicates are testing equality of arithmetic expressions (as opposed to inequalities). Here the paths through the repetitions of the second fragment are always the same.

The T-class of programs is generated in the same manner using the fragments in Figure 21, again repeating the second fragment. The conditions of the `IF` statements involve inequalities and a logical `OR`. Furthermore, this class of programs involves greater symbolic evaluation than the F-class, as the program variables get updated continually (for example, $b := b + 1$) whereas in the F-class, the variables are assigned constant numeric values (for example, $n := 5$). Here the paths through the repetitions of the second fragment alternate for each repetition; with $b > c$ true and $(b > c)$ `OR` $(x > y)$ false first, and then vice-versa.

The SN-class of programs are generated from the SN-Style program in Figure 21

by inserting multiple copies of the middle program fragment into the `THEN` branch of the previous `IF` statement, and adding an appropriate number of `FI`s in the third fragment. This produces an arbitrarily large nesting of `IF` statements.

The NSN-class is generated from the SN-Style program in Figure 21 in exactly the

```
{(x>y) AND (y>z)};
```

```
IF (x>y) AND (y>z)
THEN IF ( x>z )
     THEN  x:=x+1;
             y:=y+1;
             z:=z+1
     ELSE  a:=2
     FI
ELSE IF (x<=y) OR (y<=z)
     THEN  x:=x+1;
           y:=y+1;
           z:=z+1
     ELSE  a:=4
     FI
FI;
```

Fig. 22. Fragment for generating SSC and NSSC classes

same way except the initial fragment is excluded. The difference between these two program classes is that, in NSN, no simplification is possible using any conditioning process, whereas, in SN class of programs, the path through the program is uniquely determined.

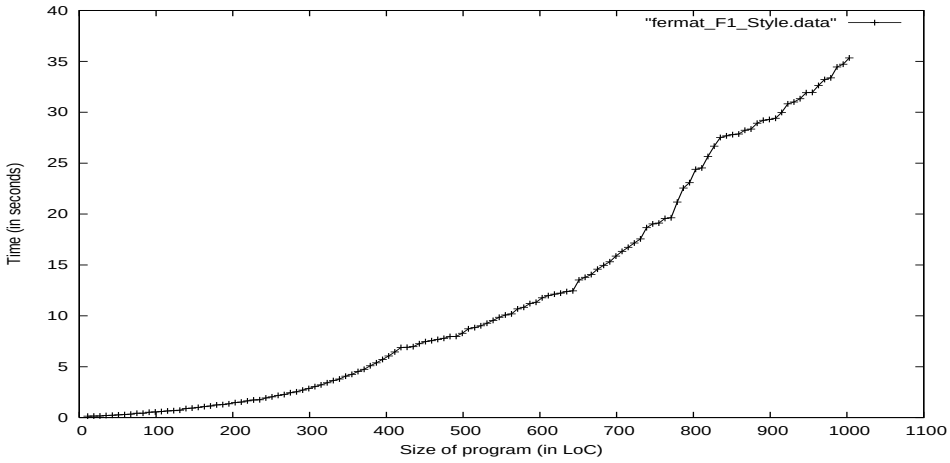
The SSC-class of programs are generated from the fragments shown in Figure 22 with multiple repetitions of the second fragment. The NSSC class is formed in exactly the same way except the initial fragment is excluded. The results of running *ConSUS* on a set of programs from each class are shown in Figures 23, 24, 25, 26, 27 and 28. These results were obtained on a Dual Pentium III with 2×330MHz and 512MB RAM running Linux. The graphs show the time taken in seconds by *ConSUS* to condition a program of a given class, plotted against the size of the program in lines of code.

Least squares regression was performed on the data sets for the following models:

- linear model $y = a + bx$;
- exponential model $y = ae^{bx}$;
- power law model $y = ax^b$;
- quadratic model $y = a + bx + cx^2$.

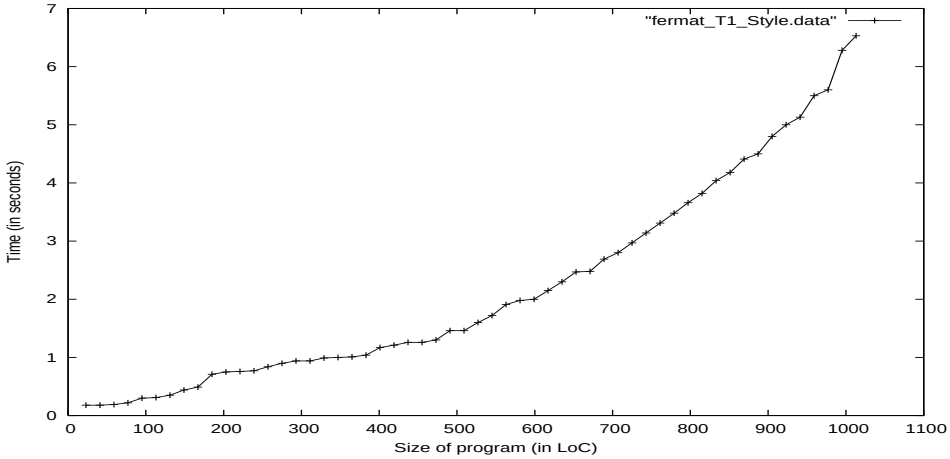
The quadratic model (with two degrees of freedom) gave the best fit to the data. The other models were significantly worse even for models of one degree of freedom. The least squares quadratic polynomials are given below each figure along with the coefficient of determination R^2.

Conditioning and conditioned slicing are typically applied to programs at the unit level, for example, as a support for detailed understanding [8], as a unit level testing aid [31] or as a unit level reuse and code extraction tool [43,42,6]. For these applications, quadratic performance is acceptable and the technique therefore appears to scale well, at least at the unit level.



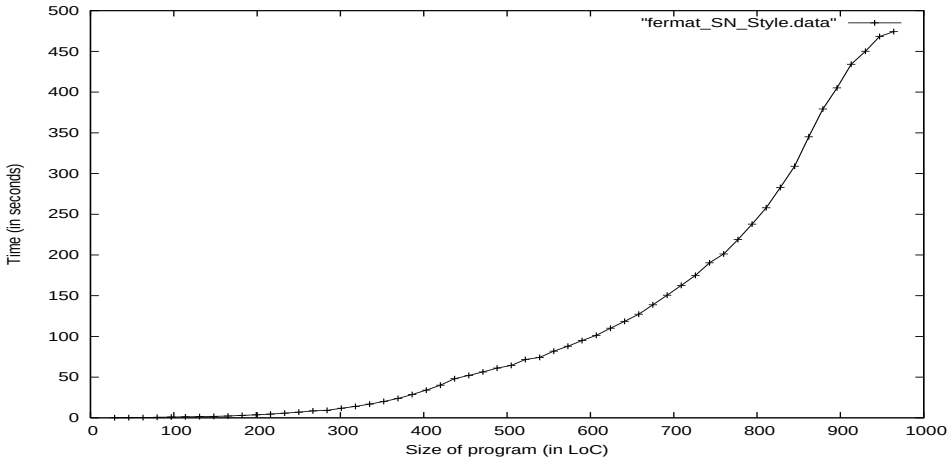
Using *FermaT*– Least squares quadratic polynomial:
$y = 6.5956 \times 10^{-1} - 4.8090 \times 10^{-3}x + 4.0246 \times 10^{-5}x^2$ with $R^2 = 0.99422$.

Fig. 23. Performance for F-class programs



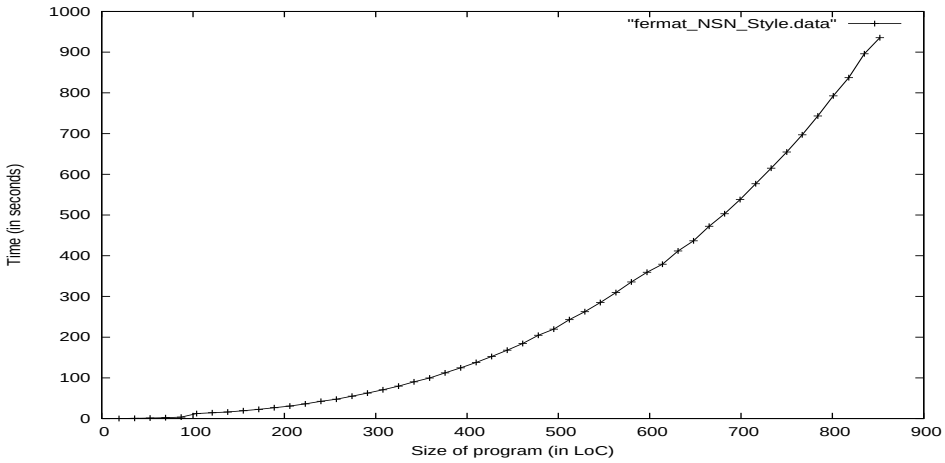
Using *FermaT* – Least squares quadratic polynomial:
$y = 4.7372 \times 10^{-1} - 1.2072 \times 10^{-3}x + 6.6462 \times 10^{-6}x^2$ with $R^2 = 0.99155$.

Fig. 24. Performance for T-class programs



Using *FermaT*– Least squares quadratic polynomial:
$y = 4.2429 \times 10^{1} - 4.1438 \times 10^{-1}x + 8.7712 \times 10^{-4}x^2$ with $R^2 = 0.98129$.

Fig. 25. Performance for SN-class programs



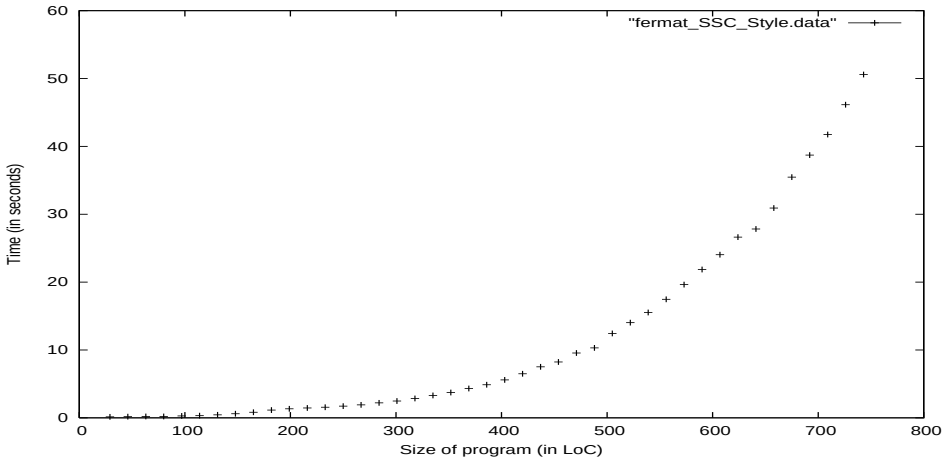
Using *FermaT* – Least squares quadratic polynomial:
$y = 4.3645 \times 10^{1} - 5.0660 \times 10^{-1}x + 1.7750 \times 10^{-3}x^2$ with $R^2 = 0.99652$.

Fig. 26. Performance for NSN-class programs

6 Conclusions

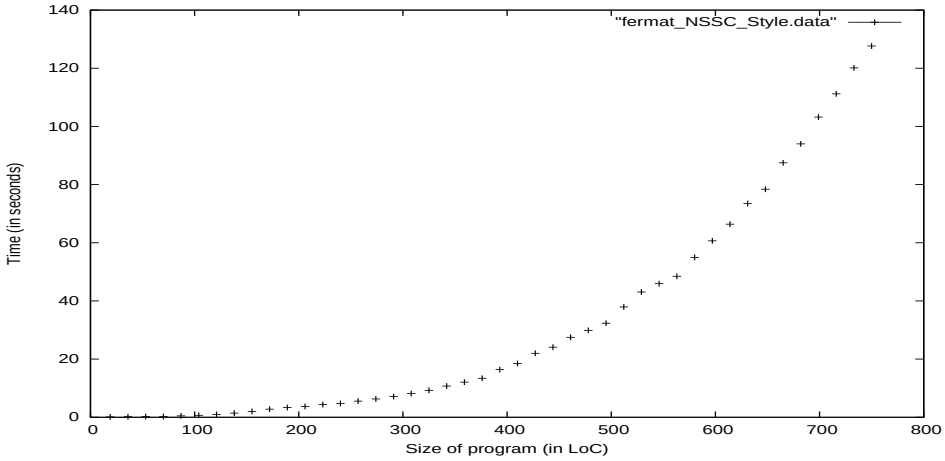
The main contributions of this paper are:

(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.
   (b) The *ConSUS* algorithm when used in conjunction with WSL's *FermaT* `Simplify` has the potential for 'scaling up' for use on larger systems.



Using *FermaT* – Least squares quadratic polynomial:
$y = 15.9617 - 0.1498x + 2.79 \times 10^{-4}x^2$ with $R^2 = 0.977$.

Fig. 27. Performance for SSC-class programs



Using *FermaT* –Least squares quadratic polynomial:
$y = 15.6851 - 0.1811x + 4.48 \times 10^{-4}x^2$
with $R^2 = 0.988$.

Fig. 28. Performance for NSSC-class programs

The algorithm defined in this paper is at the heart of *ConSUS*, a light-weight program conditioner for WSL. *ConSUS* prunes symbolic execution paths based on the validity of path conditions, thereby removing unreachable code. Unlike previous approaches, 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. Furthermore, the reasoning is achieved, not using theorem proving, but rather using the in-built expression simplifier of *FermaT*. This is a super-lightweight approach that may be capable of scaling to large programs.

In the worst case, there is no doubt that the time to perform 'best possible' program conditioning will be exponential in the size of program being conditioned. This fact is inherent to the problem of conditioning. As in the case of theorem provers, this worst case scenario does not imply that implementations of conditioners are infeasible. The reasons for this are twofold:

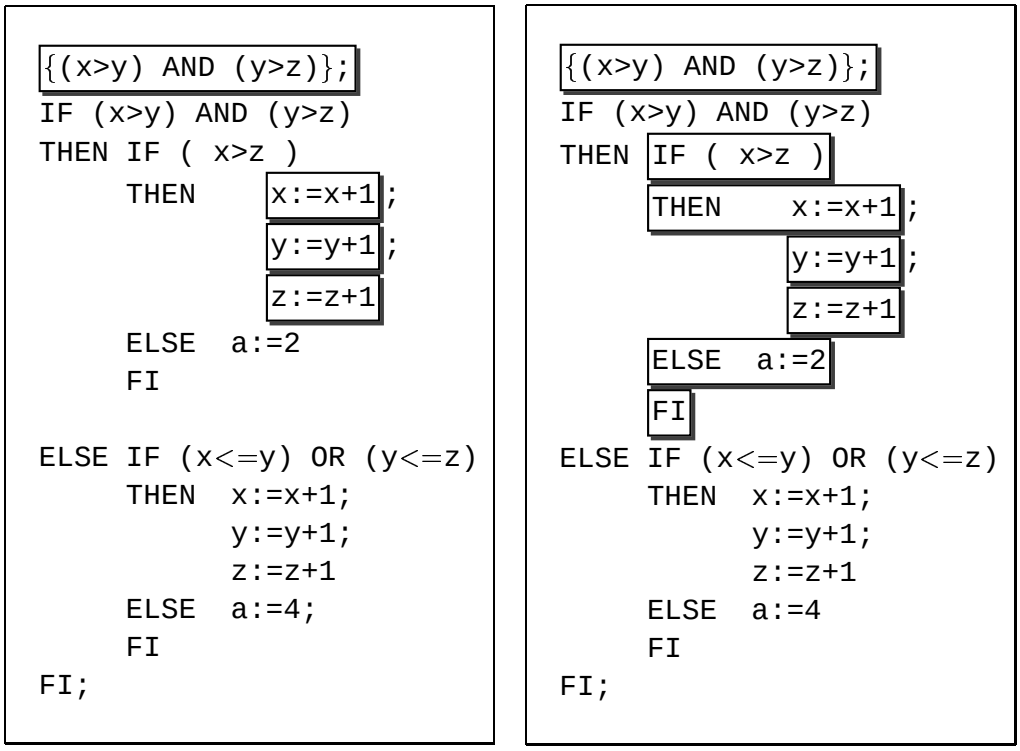
(1) In conditioning, *any* resulting reduction in program size represents progress. Even if the best possible results require exponential time, it is possible that significant reductions will be performed more quickly.
(2) In many cases conditioning may be performed in low order polynomial or even quadratic time as suggested by the empirical study in this paper. In such cases conditioning may be applicable to unit level applications at least.

A 'budget' system similar to that used by *FermaT* `Simplify` (Section 4.1.2) is envisaged whereby conditioning is performed relative to a budget. The budget is reduced as processing takes place. When the budget expires further conditioning ceases.

Clearly more work is required to scale the application of conditioning to larger programs. However, it should be stressed that conditioning is inherently harder than traditional static slicing due to the requirement of symbolic execution and theorem proving and so the authors believe that these initial results on small programs are encouraging, going some way to demonstrating the 'proof of concept' for conditioned slicing.

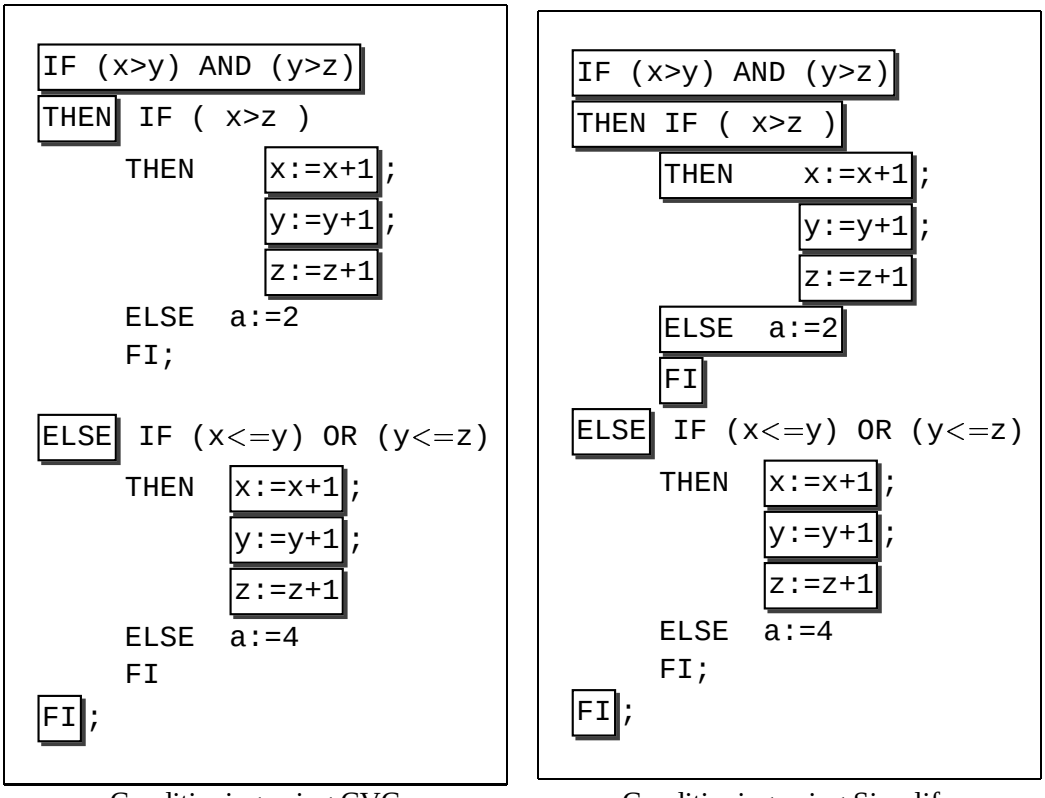
It is possible that the performance achieved using a more powerful theorem prover, in some cases may outweigh the light-weight approach when combined with 'pruning on the fly'. This is because powerful reasoning power may result in 'early pruning' which may be missed by a less powerful theorem prover. An example of a more powerful theorem prover is the Co-operating Validity Checker(CVC) [49], the successor to the Stanford Validity Checker (SVC) [65]. CVC is a high performance system for checking the validity of formulæ in a relatively rich decidable logic. CVC is applicable to boolean expressions made from these atoms.

Very recently, CVC has also been incorporated into *ConSUS*. Early results do, indeed, show that the program in Figure 6 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 $\geq$. Figure 30 illustrates the result of conditioning the second code fragment in Figure 22 using both FermaT and CVC. Figure 29 illustrates the result of conditioning the code fragments in Figure 22 using both FermaT and CVC. Future work will investigate the resulting trade off between speed and precision. Future work will also investigate whether it is possible to harness more of the power of a theorem prover like CVC in conditioning programs, for example performing induction on loop invariants. Other areas of interest include the development of heuristics that allow both



Conditioning using CVC Conditioning using Simplify

Fig. 29. Conditioning The SSC-Style program



Conditioning using CVC Conditioning using Simplify

Fig. 30. Conditioning The NSSC-Style program

heavy and light weight approaches to be combined within the same conditioner and applications of this approach to backward conditioning [33].

References

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

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

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

[4] A. P. Ershov, On the essence of computation, North–Holland, 1978, pp. 391–420.

[5] Y. Futamura, Partial evaluation of computation process – an approach to a compiler compiler, Systems, Computers, Controls 2 (5) (1971) 721–728.

[6] G. Canfora, A. Cimitile, A. De Lucia, G. A. D. Lucca, Software salvaging based on conditions, in: International Conference on Software Maintenance (ICSM'96), IEEE Computer Society Press, Los Alamitos, California, USA, Victoria, Canada, 1994, pp. 424–433.

[7] J. Field, G. Ramalingam, F. Tip, Parametric program slicing, in: 22^{nd} ACM Symposium on Principles of Programming Languages, San Francisco, CA, 1995, pp. 379–392.

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

[9] G. Canfora, A. Cimitile, A. De Lucia, Conditioned program slicing, in: M. Harman, K. Gallagher (Eds.), Information and Software Technology Special Issue on Program Slicing, Vol. 40, Elsevier Science B. V., 1998, pp. 595–607.

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

[11] S. Danicic, C. Fox, M. Harman, R. M. Hierons, The ConSIT conditioned slicing system, Software Practice and Experience Accepted for publication.

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

[13] A. Coen-Porisini, F. De Paoli, SYMBAD: A symbolic executor of sequential Ada programs, in: IFAC SAFECOMP'90, London, 1990, pp. 105–111.

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

[15] J. Krinke, G. Snelting, Validation of measurement software as an application of slicing and constraint solving, in: M. Harman, K. Gallagher (Eds.), Information and Software Technology Special Issue on Program Slicing, Vol. 40, Elsevier, 1998, pp. 661–675.

[16] R. A. DeMillo, A. J. Offutt, Experimental results from an automatic test generator, acm Transactions of Software Engineering and Methodology 2 (2) (1993) 109–127.

[17] K. N. King, A. J. Offutt, A FORTRAN language system for mutation-based software testing, Software Practice and Experience 21 (1991) 686–718.

[18] A. J. Offutt, An integrated system for automatically generating test data, in: R. T. Ng, Peter A.; Ramamoorthy, C.V.; Seifert, Laurence C.; Yeh (Ed.), Proceedings of the First International Conference on Systems Integration, IEEE Computer Society Press, Morristown, NJ, 1990, pp. 694–701.

[19] B. Korel, J. Rilling, Dynamic program slicing methods, in: M. Harman, K. Gallagher (Eds.), Information and Software Technology Special Issue on Program Slicing, Vol. 40, Elsevier, 1998, pp. 647–659.

[20] F. Tip, A survey of program slicing techniques, Journal of Programming Languages 3 (3) (1995) 121–189.

[21] D. W. Binkley, K. B. Gallagher, Program slicing, in: M. Zelkowitz (Ed.), Advances in Computing, Volume 43, Academic Press, 1996, pp. 1–50.

[22] A. De Lucia, Program slicing: Methods and applications, in: 1^{st} IEEE International Workshop on Source Code Analysis and Manipulation, IEEE Computer Society Press, Los Alamitos, California, USA, Florence, Italy, 2001, pp. 142–149.

[23] M. Harman, R. M. Hierons, An overview of program slicing, Software Focus 2 (3) (2001) 85–92.

[24] D. W. Binkley, M. Harman, A survey of empirical results on program slicing, Advances in Computers To appear.

[25] M. Harman, R. M. Hierons, S. Danicic, J. Howroyd, C. Fox, Pre/post conditioned slicing, in: IEEE International Conference on Software Maintenance (ICSM'01), IEEE Computer Society Press, Los Alamitos, California, USA, Florence, Italy, 2001, pp. 138–147.

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

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

[28] D. W. Binkley, The application of program slicing to regression testing, in: M. Harman, K. Gallagher (Eds.), Information and Software Technology Special Issue on Program Slicing, Vol. 40, Elsevier, 1998, pp. 583–594.

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

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

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

[32] D. W. Binkley, M. Harman, L. R. Raszewski, 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), IEEE Computer Society Press, Los Alamitos, California, USA, Limerick, Ireland, 2000, pp. 161–170.

[33] C. Fox, M. Harman, R. M. Hierons, 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), IEEE Computer Society Press, Los Alamitos, California, USA, Toronto, Canada, 2001, pp. 89–97.

[34] J. Rilling, A.Seffah, J.Lukas, MOOSE – a software comprehension framework, in: 5^{th} World Multi-Conference on systemics, cybernetics and informatics (SCI 2001), to appear.

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

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

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

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

[39] L. M. Ott, J. J. Thuss, The relationship between slices and module cohesion, in: Proceedings of the 11^{th} ACM conference on Software Engineering, 1989, pp. 198–204.

[40] H. D. Longworth, L. M. Ott, M. R. Smith, The relationship between program complexity and slice complexity during debugging tasks, in: Proceedings of the Computer Software and Applications Conference (COMPSAC'86), 1986, pp. 383–389.

[41] G. Canfora, A. Cimitile, M. Munro, RE^2: Reverse engineering and reuse re-engineering, Journal of Software Maintenance : Research and Practice 6 (2) (1994) 53–72.

[42] A. Cimitile, A. De Lucia, 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), IEEE Computer Society Press, Los Alamitos, California, USA, Nice, France, 1995, pp. 124–133.

[43] A. Cimitile, A. De Lucia, M. Munro, Qualifying reusable functions using symbolic execution, in: Proceedings of the 2^{nd} working conference on reverse engineering, IEEE Computer Society Press, Los Alamitos, California, USA, Toronto, Canada, 1995, pp. 178–187.

[44] Grammatech Inc., The codesurfer slicing system (2002).
URL `www.grammatech.com`

[45] D. W. Binkley, M. Harman, A large-scale empirical study of forward and backward static slice size and context sensitivity, in: IEEE International Conference on Software Maintenance (ICSM 2003), IEEE Computer Society Press, Los Alamitos, California, USA, Amsterdam, Netherlands, 2003, to Appear.

[46] L. Ouarbya, S. Danicic, D. M. Daoudi, M. Harman, C. Fox, A denotational interprocedural program slicer, in: IEEE Working Conference on Reverse Engineering (WCRE 2002), IEEE Computer Society Press, Los Alamitos, California, USA, Richmond, Virginia, USA, 2002, pp. 181 – 189.

[47] P. A. Hausler, Denotational program slicing, in: 22^{nd}, Annual Hawaii International Conference on System Sciences, Volume II, 1989, pp. 486–495.

[48] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an efficient SAT solver, in: 39th Design Automation Conference, 2001, las Vegas.

[49] A. Stump, C. W. Barrett, D. L. Dill, CVC: a cooperating validity checker, in: J. C. Godskesen (Ed.), Proceedings of the International Conference on Computer-Aided Verification, Lecture Notes in Computer Science, 2002.

[50] L. Zhang, S. Malik, The quest for efficient boolean satisfiability solvers, in: Proceedings of 14th Conference on Computer Aided Verification (CAV 2002), 2002, copenhagen, Denmark.

[51] M. Ward, Proving program refinements and transformations, DPhil Thesis, Oxford University (1989).

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

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

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

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

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

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

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

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

[60] M. J. Harrold, N. Ci, Reuse-driven interprocedural slicing, in: Proceedings of the 20th International Conference on Software Engineering, IEEE Computer Society Press, 1998, pp. 74–83.

[61] D. Liang, M. J. Harrold, Reuse-driven interprocedural slicing in the presence of pointers and recursion, in: IEEE International Conference of Software Maintenance, IEEE Computer Society Press, Los Alamitos, California, USA, Oxford, UK, 1999, pp. 410–430.

[62] M. Mock, D. C. Atkinson, C. Chambers, S. J. Eggers, Improving program slicing with dynamic points-to data, in: W. G. Griswold (Ed.), Proceedings of the 10^{th} ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-02), ACM Press, New York, 2002, pp. 71–80.

[63] A. Nishimatsu, M. Jihira, S. Kusumoto, K. Inoue, Call-mark slicing: An efficient and economical way of reducing slices, in: Proceedings of the 21st International Conference on Software Engineering, ACM Press, 1999, pp. 422–431.

[64] Calendar Act, Calendar Act, Anno vicesimo quarto George II, cap. xxiii. (1751).

[65] C. Barrett, D. Dill, J. Levitt, Validity checking for combinations of theories with equality, in: M. Srivas, A. Camilleri (Eds.), Formal Methods In Computer-Aided Design, Vol. 1166 of Lecture Notes in Computer Science, Springer-Verlag, 1996, pp. 187–201, palo Alto, California, November 6–8.