ConSUS :
A Scalable Approach to Conditioned Slicing

M. Daoudi, L. Ouarbya, J. Howroyd & S. Danicic
Goldsmiths College
University of London
London SE14 6NW
United Kingdom

Mark Harman
Brunel University
Uxbridge
Middlesex
UB8 3PH
United Kingdom

Chris Fox
University of Essex
Wivenhoe Park
Colchester
CO4 3SQ
United Kingdom

M. P. Ward
Software Technology Research Lab
De Montfort University
The Gateway,
Leicester LE1 9BH
United Kingdom

Keywords: Conditioned Slicing, FermaT, Program Comprehension.

Abstract

Conditioned slicing can be applied to reverse engineering problems which involve the extraction of executable fragments of code in the context of some criteria of interest. This paper introduces ConSUS*, a conditioner for the Wide Spectrum Language, WSL. The symbolic executor of* ConSUS *prunes the symbolic execution paths, and its predicate reasoning system uses the FermaT* `simplify` *transformation in place of a more conventional theorem prover. We show that this combination of pruning and simplification-as-reasoner leads to a more scalable approach to conditioning.*

1 Introduction

Program slicing is a source code extraction technique that allows a reverse engineer to extract an executable subprogram based upon a slicing criterion. The original formulation of slicing [24] was static. That is, the slicing criterion contained no information about the input to the program. Later work on slicing created different paradigms for slicing including dynamic slicing [1, 17] (for which the input is known) and quasi-static slicing [19] (for which an input prefix is known).

The way in which slicing produces an executable subprogram, based upon some criterion of interest, gives rise to applications in re-engineering. For example, slicing has been suggested as a tool for the integration of two different versions of a program [16]. It also forms part of approaches to decompilation [5, 6] and has been put forward as part of a tool-assisted approach to program comprehension [2, 10, 12].

This paper is concerned with a variation of slicing called conditioned slicing[1] [3, 13]. 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.

Conditioned slicing further extends the applicability of traditional slicing to problems in reverse engineering, because the additional ability to express conditions allows the reverse engineer to refine the code extraction to conditions of interest. For example, Canfora et al. [4] show how a form of conditioning can be used to isolate reusable functions from large monolithic chunks of code. De Lucia et al. [10] show how conditioned slicing can be used as part of an approach to the initial code comprehension which typically precedes reverse engineering tasks. Cimitile et al. [8, 7] show that conditioned slicing and related techniques can be used to extract and reuse functions during reverse and re-engineering.

The conditioned slicing criterion is a triple, $(\pi,\ V,\ n)$ where π is some condition of interest and $(V,\ n)$ are the two components of the static slicing criterion. In this paper, we shall be concerned with the conditioning phase of conditioned slicing, and so the criterion of interest is simply some

[1] A similar approach called constrained slicing was introduced by Field et al. [11].

condition. Where no condition is given, the system will thus simply attempt to remove infeasible paths (a useful step in itself). The paper introduces the *ConSUS* conditioning system, which is implemented for the Wide Spectrum Language, WSL [21]. WSL is the language used in the FermaT Transformation system [20] and which has been previously used as part of a transformation-based approach to reverse engineering [23]. We chose WSL to allow us to combine our work on conditioning with our work on slicing [22] and amorphous slicing [12, 14, 18]. This will (ultimately) allow us to produce an amorphous conditioned slicer.

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. Space prevents a full explanation of the WSL syntax and semantics.

As an example (both of WSL and of the way in which conditioning identifies sub-programs) consider the Taxation program in Figure 1. The figure contains a fragment[2] 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 reverse 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.

After conditioning a program, a conditioned slice can be obtained by applying static slicing to the conditioned program. For example, the conditioned slice on the variable `tax` for the condition above is depicted in Figure 1, by shading the lines of the conditioned program which are identified by static slicing.

This paper describes the *ConSUS* system, focusing upon its approach to symbolic execution and to determining the outcome of symbolic predicates. These two features have been designed to allow the technique to scale more readily to larger systems. The principal contributions of this paper are

- An approach to symbolic execution is used which exploits the simplification embodied in conditioning to prune symbolic paths before they are created, speeding up the analysis.
- A new implementation for reasoning about symbolic states and path conditions is introduced, which uses the FermaT `simplify` transformation to decide propositions.
- Initial empirical results are presented which show that the approach scales reasonably well (our experiments fit quadratic curves).

The rest of this paper is organized as follows. Section 2 introduces an integrated approach to symbolic execution which combines conditioning and symbolic execution to prune paths as the symbolic execution proceeds. Section 3 describes our use of the FermaT `Simplify` transformation to achieve a form of super-lightweight theorem proving, which is required to determine the outcome of symbolic predicates in a symbolic conditioned-state pair. Section 4 presents the results of an empirical investigation into the performance of the approach and section 5 concludes with directions for further work.

2 Symbolic Execution

The *ConSUS* tool uses a three phase approach:

1. Symbolically Execute: to propagate assertions through the program where possible;
2. Produce a Conditioned Program: eliminate statements which are never executed under the given condition;
3. Perform Static Slicing

Steps 1 and 2 are integrated into a single symbolic executor and conditioner. This allows the conditioning to prune the execution traces to be considered for symbolic execution. The slicer we use handles side effects across procedure calls, but the details are beyond the scope of this paper[3].

For conditioning a program, we need not only a symbolic state, but a set of path conditions which represents the sequences of conditions which must be true in order for a given symbolic state to pertain at a given point. These symbolic paths are built up as the symbolic executor moves through the program. Because there are typically several feasible paths, the overall symbolic state, which we call $Bigstate$, contains a set of pairs. Each pair consists of a symbolic path condition and a symbolic state.

More formally, a $Bigstate$ for a program P is defined as:

$$\{(\sigma_i, \pi_i)\}_{i=1}^{n}$$

where each pair (σ_i, π_i) corresponds to a conditioned-state. π_i is a boolean expression representing the conditions under

[2]This is WSL version of the C program previously used in [9].

[3]Note to Referee: A separate paper has been submitted to WCRE which describes the slicer and the way in which it handles side effects.

```
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 ;
     IF (income <= pc10) THEN tax := income * rate10
     ELSE tax := pc10 * rate10 ;
          income := income - pc10 ;
          IF (income <= 28000) THEN tax := tax + income * rate23
          ELSE tax := tax + 28000 * rate23 ;
               income := income - 28000 ;
               tax := tax + income * rate40 FI FI FI;
```

Key

Original Program: Unboxed lines of code

Conditioned program: boxed lines of code

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

Conditioned slice: shaded boxed lines of code

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

which a possible path to a statement is taken. σ_i represents the symbolic state of the variables on that path. We think of σ_i as a function that maps program variables into their symbolic value:

$$\sigma_i\colon Variables \to Expressions$$

The variables in the program will have the symbolic values given in the first element of the pair if the conditions in the second element are true.

The symbolic execution starts with an empty symbolic state, and a null path condition, which can be interpreted as the universally valid true proposition.

In order to condition a program P, we define the following functions :

- $Condition\colon Program \times Bigstate \to Program$
- $BigUpdate\colon Program \times Bigstate \to Bigstate$
- $Update\colon Program \times Pair \to Bigstate$
- $eval\colon Expression \times state \to Expression$
- $prove\colon Boolean\_Expression \to \{T, F, \bot\}$

The function $Condition$, takes a program, p and a $Bigstate$, Σ, and produces the program which results from conditioning p with respect to Σ. Thus $Condition$ is the top level function which is used to condition a program. The function $BigUpdate$ takes a program, p and a $Bigstate$, Σ, and returns the new $Bigstate$ that results from symbolically executing p in $Bigstate$. The function $BigUpdate$ will be defined in terms of a function over individual conditioned-state pairs, called $Update$. In order to define $Condition$ and $BigUpdate$, we require two auxiliary functions $eval$ and $prove$. $eval$ takes and expression, e and a state and returns the expression which results from evaluating e in the symbolic state, σ. This is obtained by substituting variables mentioned in e for the symbolic values they denote in σ. The function $prove$ denotes the theorem prover at the heart of the conditioner. This is a detachable component of the conditioner. *ConSUS* uses the FermaT `simplify` transformation to implement a super-lightweight theorem prover, in a manner described in the next section. In this section the $prove$ function will be treated as a black box, which takes a symbolic boolean expression over inequalities between integer arithmatic expressions b and returns one of three possible values. The returned value T, indicates that b can be proved to be true. The returned value F, indicates that b can be proved to be false. The returned value $\bot$, indicates that b can be proved neither to be true nor to be false.

Of course, the fact that some boolean expression b can be proved to be neither true nor false does not provide any information and there will be some provable booleans which our system (and which *any* conceivable replacement) will fail to decide. However, the approach will be safe, so that $prove$ will correctly decide a subset of those statements which are tautologies and contradictions.

Also for each statement S, the set of post conditioned-state's corresponding to a set $\{(\sigma_i, \pi_i)\}_{i=1}^{n}$ of prior conditioned-state's is formed by union as :

$$BigUpdate(S, \{(\sigma_i, \pi_i)\}_{i=1}^{n}) = \bigcup_{i=1}^{n} Update(S,\ (\sigma_i, \pi_i))$$

We will describe the conditioning of a subset of WSL, which includes sufficient features to explain the approach.

2.1 Conditioning A Sequence Of Statements

Conditioning a sequence of statements $S_1;\ S_2$ is defined in the following two steps:

$$BigUpdate(S_1;\ S_2, Bigstate)\ =_{\text{DF}}\ BigUpdate(S_2, R)$$

$$Condition(S_1;\ S_2, Bigstate)$$
$$=_{\text{DF}}\ \ Condition(S_1, Bigstate);\ Condition(S_2, R)$$

where :
$R = BigUpdate(S_1, Bigstate)$

2.2 Conditioning An Assignment Statement

Let $S\ =_{\text{DF}}\ v := e$.
The $Condition$ function is defined as follows:

$$Condition(v :=\ e\ ,\ Bigstate)\ =_{\text{DF}}\ v := e$$

The set $Update(x := e, (\sigma_i, \pi_i))$ of post conditioned-state pairs corresponding to a prior conditioned-state (σ_i, π_i) is formed by adding to each symbolic state the fact that the variables on the left-side of the assignment statement is bound to the value of the expression on the right side, where all variables occurring in the expression are replaced by their current symbolic values given in the respective state. Any variable occurring in the expression which does not already have a symbolic value in the relevant state is assigned a unique symbolic constant value (rather like a skolem constant).

The $Update$ function is defined as follows:

$$Update(v{:=}e, (\sigma_i, \pi_i))\ =_{\text{DF}}\{(\sigma_i \oplus \{x \to eval(e, \sigma_i)\}, \pi_i)\}$$

where $\oplus$, is the assignment function update function defined below

$$f \oplus g \;=_{\text{DF}} \{\langle v, e\rangle : \langle v, e\rangle \in g \vee \langle v, e\rangle \in f \wedge \neg\exists e'.\langle v, e'\rangle \in g)\}$$

2.3 Conditioning An `IF` Statement

Let $S \;=_{\text{DF}}$ if B then S_1 else S_2 fi.

Given a set $\{\ (\sigma_i, \pi_i)\ \}_{i=1}^{n}$ of prior conditioned-state's, the post conditioned-state's and the conditioned statement are now defined.
Define for each new path condition π_i, from a conditioned-state (σ_i, π_i) say, the true and false path conditions π_i^T and π_i^F, respectively, by

$$\pi_i^T \;=_{\text{DF}}\; \pi_i \wedge eval(B, \sigma_i)$$

$$\pi_i^F \;=_{\text{DF}}\; \pi_i \wedge eval(\neg B, \sigma_i)$$

Then
A set Update(S,(σ_i, π_i)) of post conditioned-state's corresponding to a prior conditioned-state (σ_i, π_i) is given according to whether

$$c_i \;=_{\text{DF}}\; \pi_i \Rightarrow eval(B, \sigma_i)$$

is provably true or false or otherwise as follows:

- If prove$(c_i) = T$ then
  Update(S,(σ_i, π_i)) $=_{\text{DF}}$ Update(S_1,(σ_i, π_i^T))
- If prove$(c_i) = F$ then
  Update(S,(σ_i, π_i)) $=_{\text{DF}}$ Update(S_2,(σ_i, π_i^F))
- Otherwise

$$Update(\mathrm{S}, (\sigma_i, \pi_i)) \\ =_{\text{DF}} \quad Update(S_1, (\sigma_i, \pi_i^T)) \cup Update(S_2, (\sigma_i, \pi_i^F))$$

$Condition(S, \{(\sigma_i, \pi_i)\}_{i=1}^{n})$ is defined according to whether for all $i = 1, 2 \ldots, n$, it is the case that c_i is provably true, or provably false or neither, as follows :

- If prove$(c_i) = T$ for all i then

$$Condition(\mathrm{S}, \{(\sigma_i, \pi_i)\}_{i=1}^{n}) \\ =_{\text{DF}} \quad Condition(S_1, \{\ (\sigma_i, \pi_i^T)\ \}_{i=1}^{n})$$

- If prove$(c_i) = F$ for all i then

$$Condition(\mathrm{S}, \{(\sigma_i, \pi_i)\}_{i=1}^{n}) \\ =_{\text{DF}} \quad Condition(S_2, \{\ (\sigma_i, \pi_i^F)\ \}_{i=1}^{n})$$

- Otherwise

$$Condition(\mathrm{S}, \{(\sigma_i, \pi_i)\}_{i=1}^{n}) \\ =_{\text{DF}} \quad if\ B\ then\ S_1'\ else\ S_2'\ fi$$

  where :
  $S_1' = Condition(S_1, \{\ (\sigma_i, \pi_i^T)\ \}_{i=1}^{n})$
  $S_2' = Condition(S_2, \{\ (\sigma_i, \pi_i^F)\ \}_{i=1}^{n})$

Original program P	Conditioned program P'

<pre>x := 0 ;
y := 1 ;
if x = 0
 then **if** y = 2
          then n := 3;
          else n := 4;
      fi;
 else n := 5;
fi;
if n = 3
   then a := 5;
   else a := 7;
if ;</pre>
	<pre>x := 0 ;
	y := 1 ;
	
	          n := 4;
	
	          a := 7;
	</pre>

Figure 2. Conditioning an `IF` statement

Figure 2 illustrates a basic example of this analysis using a four-statement program containing two assignments followed by two `IF` statements.
The set of conditioned-states ($Bigstate$) after the execution of the two initial assignments is $\{([(x,0),(y,1)], True)\}$. At the first predicate $True \Rightarrow 0 = 0$ is sent to the simplifier and is simplified to True. As a result, the false branch of the `IF` statement is removed and $Bigstate$ becomes $\{([(x,0),(y,1)], True \wedge 0 = 0)\}$. At the nested predicate $True \wedge 0 = 0 \Rightarrow 1 = 2$ is sent to the simplifier and is simplified to False. As a result, the true branch of the nested `IF` statement is removed and $Bigstate$ becomes $\{([(x,0),(y,1),(n,4)], True \wedge 0 = 0 \wedge \neg(1 = 2))\}$. Finally $True \wedge 0 = 0 \wedge \neg(1 = 2)) \Rightarrow 4 = 3$ is simplified to False resulting, in the removal of the true part of the `IF` program statement, and $Bigstate$ becoming $\{([(x,0),(y,1),(n,4),(a,7)], True \wedge 0 = 0 \wedge \neg(1 = 2)) \wedge \neg(4 = 3)\}$

2.4 Conditioning A `WHILE` Statement

Let

$$S \;=_{\text{DF}}\; \text{while B do } S_0 \text{ od}$$

and

$$c_i \;=_{\text{DF}}\; \pi_i \Rightarrow eval(B, \sigma_i)$$

There are three cases to consider :

- If prove$(c_i) = F$ then

$$Update(S, (\sigma_i, \pi_i)) \;=_{\text{DF}}\; \{(\sigma_i, \pi_i^F)\}$$

A negation of the the condition B is added to a copy of each of the current conditioned-state pairs, replacing all variables by their symbolic values in the corresponding symbolic states.

- If prove$(c_i) = T$ then

$$Update(S,(\sigma_i,\pi_i)) \ =_{\text{DF}}$$
$$Update(S,(\sigma_i',\pi_i \wedge eval(B,\sigma_i') \wedge \neg eval(B,\sigma_i'')))$$

where :
σ_i' : is the symbolic state at the start of the final iteration.
σ_i'' : is the symbolic state obtained after the final iteration.

The loop body S_0 is executed at least once. If the loop terminates, then at the final execution of S_0 the loop may or may not have already been executed. In general it will have been executed several times before. It is not easy to obtain precise symbolic representations of any variables which might have been assigned values during previous iterations of the loop.
The approach proceeds as follows:

1. To a copy of all the prior conditioned-state pairs is added the fact that the condition B is initially true.
2. Conceptually, S_0 is then symbolically executed just once, in the context of the conditioned-state pairs that result, except that any variables which might have been assigned values in previous iterations around the loop are treated as if they have previously been assigned values that are unique symbolic constants.
3. As the loop condition must have been true at the start of the start of the final iteration, and false following the final iteration, to each of the conditioned-state pairs that result from the final iteration, the algorithm adds the loop condition, as evaluated in the symbolic states in the beginning of the final iteration, and the negation of the loop condition as evaluated at the end of the final iteration.

- Otherwise
The union of the conditioned-state's that result from the previous two cases is formed.

The Conditioned statement

$$Condition(S,\{(\sigma_i,\pi_i)\}_{i=1}^{n})$$

is defined according to whether for all $i = 1, 2, \ldots, n$; c_i is provably false or otherwise, as follows :

Original program P	Conditioned program P'
`x := 0;`	`x := 0;`
`y := 2;`	`y := 2;`
while `x < 0` **do**	
`y := 1;`	
`x := x−1;`	
od;	
if `y = 2;`	
then `n := 4;`	`n := 4;`
else `n := 5;`	
fi;	
while `n > 2` **do**	**while** `n > 2` **do**
`@PRINT(n);`	`@PRINT(n) ;`
`n := n − 1;`	`n := n − 1;`
od;	**od**:
if `n > 2`	
then `a:=6;`	
else `a:=7;`	`a := 7;`
fi;	

Figure 3. Conditioning a `WHILE` Loop

- If $prove(c_i) = F$ for all i then

$$Condition(S,\{(\sigma_i,\pi_i)\}_{i=1}^{n}) \ =_{\text{DF}} \ skip$$

- Otherwise
$Condition(S,\{(\sigma_i,\pi_i)\}_{i=1}^{n}) \ =_{\text{DF}} \ while\ B\ do\ S_0'\ od$
where :
$S_0' = Condition(S_0, \{\ (\sigma_i',\pi_i \wedge eval(B,\sigma_i')\ \}_{i=1}^{n})$

Figure 3 illustrates the effect of conditioning on a `WHILE` loop.

2.5 Conditioning An Assert Statement

Let $S \ =_{\text{DF}} \ \{B\}$ where $\{B\}$ is an assertion statement. Such a statement acts as a partial skip statement, and can be thought of as being equivalent to

$$while\ \neg B\ do\ skip\ od$$

That is, if the condition B is true, then the statement terminates immediately without changing any variables, otherwise it fails to terminate [13]. Moreover, an assert statement is very helpful from a practical point of view as we can insert the conditioned slicing criterion directly into the program as program code. For an assert, the functions $Condition$ and $BigUpdate$ are defined as follows:

$$Update(\{\ B\ \},(\sigma_i,\pi_i))$$
$$= \begin{cases} \{\ (\sigma_i,\pi_i)\ \} & if\ \forall i\ \ prove(c_i)\ =\ T \\ \{\ (\sigma_i,F)\ \} & if\ \forall i\ \ prove(c_i)\ =\ F \\ \{\ (\sigma_i,\pi_i^T)\ \} & otherwise \end{cases}$$

$$Condition(\{\ B\ \}, Bigstate) = \begin{cases} Skip & if\, \forall i\ prove(c_i)\ =\ T \\ abort & if\, \forall i\ prove(c_i)\ =\ F \\ \{\ B\ \} & otherwise \end{cases}$$

3 Super-lightweight Theorem Proving with FermaT's `simplify` Transformation

Beside having its own parser, a major reason behind implementing a conditioned slicer in WSL was the many available transformations in FermaT. The ability to simplify conditions is important for any conditioned slicer. An obvious solution is to use an existing theorem prover; previous approaches to conditioned slicing either used this approach or suggested that it should be used [3, 9]. Unfortunately, the use of a theorem prover can impose a large overhead in both memory and CPU time.

The theorems of relevance in *ConSUS* typically involve inequalities over arithmetic expressions. There may be techniques that are more appropriate for these kinds of theorems than general purpose theorem provers. One such technique is the one adapted by the built in simplifier in the FermaT workbench.

3.1 Advantages and Disadvantages

The FermaT expression and condition simplifier's design is aimed at providing a fast and efficient simplification for common expressions and conditions which occur during transformation, while providing a fast response on expressions and conditions which cannot be easily simplified.

For scalability, the requirements for an expression and condition simplifier for the FermaT transformation system were for it to be :

1. Efficient, especially on small expressions.
2. Easily extendible. It would be difficult to attempt to simplify *all* possible expressions which are capable of simplification. 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 we can prove the correctness of the extended simplifier equally easily.

4 Empirical Validation

We considered four classes of programs F, T, SN, and NSN. The programs in each class are formed from program fragments with multiple repetitions of one of these fragments. 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 4 with multiple repetitions of the second fragment. This set of programs tests the conditioning process of ConSUS on sequential if statements where the conditions are testing equality of arithmetic expressions (as opposed to inequalities). Here the paths through the repetitions of the second fragment is always the same.

The T-class of programs is generated in the same manner using the fragments in Figure 5, 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, $b := b + 1$) where as 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-verse.

The SN-class of programs are generated from the program fragments in Figure 6 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 formed in exactly the same way except the initial fragment is excluded. The difference between these two program classes is that in NSN no simplification can be performed by *ConSUS* where as in SN the path through the programs is uniquely determined. With these two classes the performance of ConSUS is tested in the presence of nested if statements in best and worse case scenarios.

The results of running *ConSUS* on a set of programs from each class are shown in Figures 8, 7, 9 and 10. 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 $y = a + bx + cx^2$.

```
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;
```

Figure 4. F1-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;
```

Figure 5. T1-Style Program

The quadratic model (with two degrees of freedom) gave the best fit to the data. The other models being 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.

For an analysis as complex and semantically-intricate as conditioning, it is unreasonable to expect linear or near linear performance, so quadratic complexity would appear to be the best we can hope for. This is because conditioning involves theorem proving (of a kind) and symbolic execution which can be computationally expensive.

Fortunately, conditioning and conditioned slicing are typically applied to programs at the unit level, for example as a support for detailed understanding [10], as a unit level testing aid [15] or as a unit level reuse and code extraction tool [8, 7, 4]. For these applications, quadratic performance is acceptable and the technique therefore appears to scale well, at least at the unit level.

```
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;
```

Figure 6. Simplifiable Nested Program

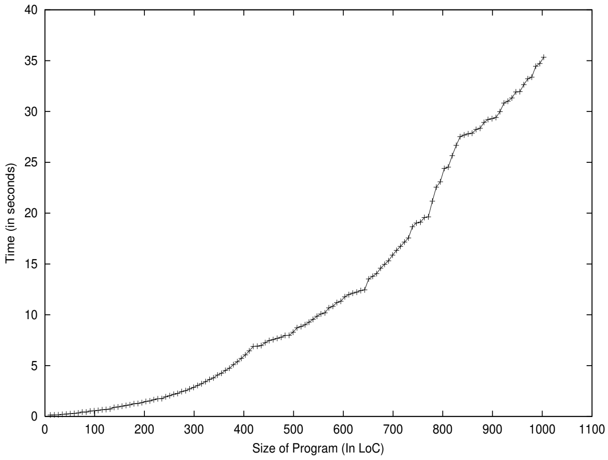

Figure 7. Performance for F-class programs

Least squares quadratic polynomial is

$$y = 6.5956 \times 10^{-1} - 4.8090 \times 10^{-3}x + 4.0246 \times 10^{-5}x^2$$

with $R^2 = 0.99422$.

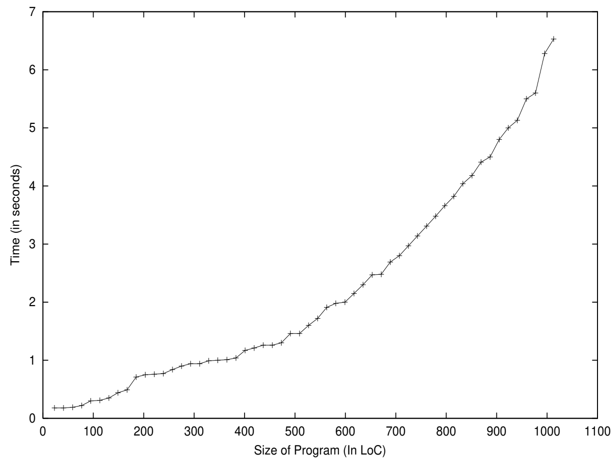

Figure 8. Performance for T-class programs

Least squares quadratic polynomial is

$$y = 4.7372 \times 10^{-1} - 1.2072 \times 10^{-3}x + 6.6462 \times 10^{-6}x^2$$

with $R^2 = 0.99155$.

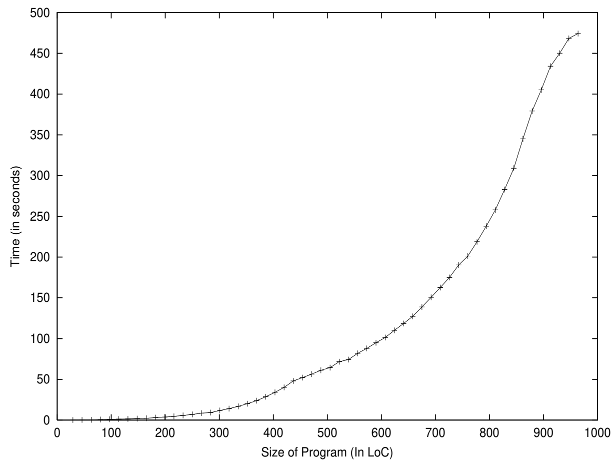

Figure 9. Performance for SN-class programs

Least squares quadratic polynomial is

$$y = 4.2429 \times 10^{1} - 4.1438 \times 10^{-1}x + 8.7712 \times 10^{-4}x^2$$

with $R^2 = 0.98129$.

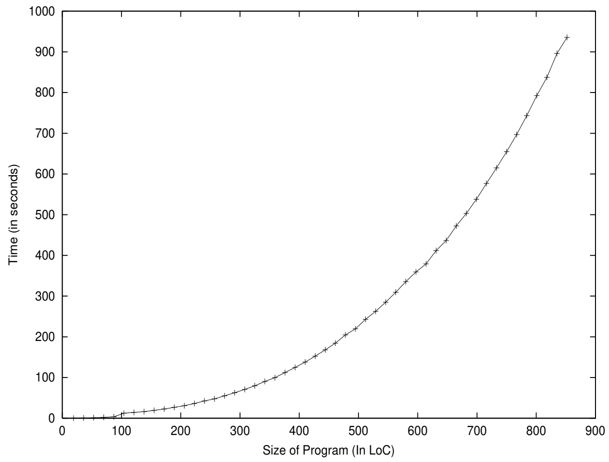

Figure 10. Performance for NSN-class programs

Least squares quadratic polynomial is

$$y = 4.3645 \times 10^{1} - 5.0660 \times 10^{-1}x + 1.7750 \times 10^{-3}x^2$$

with $R^2 = 0.99652$.

5 Conclusions

This paper has introduced a conditioner, *ConSUS*, for the Wide Spectrum Language WSL. As with previous work the approach involves both symbolic execution and reasoning about symbolic predicates to determine whether they either must be true or false given the information built up in the symbolic paths traversed.

Unlike previous approaches, the *ConSUS* system integrates the reasoning and symbolic execution within a single system. Our empirical analysis of the approach suggests that for 'reasonable' conditioning tasks, the algorithm is polynomial in the size of the program to be conditioned.

References

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

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

[3] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program*

Slicing, volume 40, pages 595–607. Elsevier Science B. V., 1998.

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

[5] C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *IEEE International Conference on Software Maintenance (ICSM'97)*, pages 188–195. IEEE Computer Society Press, Los Alamitos, California, USA, 1997.

[6] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Software Practice and Experience*, 25(7):811–829, July 1995.

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

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

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

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

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

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

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

[14] M. Harman, L. Hu, X. Zhang, and M. Munro. GUSTT: An amorphous slicing system which combines slicing and transformation. In 1^{st} *Workshop on Analysis, Slicing, and Transformation (AST 2001)*, pages 271–280, Stuttgart, Oct. 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

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

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

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

[18] Y. Sivagurunathan, M. Harman, and B. Sivagurunathan. Slice-based dynamic memory modelling – a case study. In 26^{th} *IEEE Annual Computer Software and Applications Conference (COMPSAC 2002)*, Oxford, UK, Aug. 2002. IEEE Computer Society Press, Los Alamitos, California, USA. To Appear.

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

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

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

[22] M. Ward. Program slicing via FermaT transformations. In 26^{th} *IEEE Annual Computer Software and Applications Conference (COMPSAC 2002)*, Oxford, UK, Aug. 2002. IEEE Computer Society Press, Los Alamitos, California, USA. To Appear.

[23] M. Ward and K. Bennett. A practical program transformation system. In *Working Conference on Reverse Engineering*, pages 212–221, Baltimore, MD, USA, May 1993. IEEE Computer Society Press, Los Alamitos, California, USA.

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