Pre/Post Conditioned Slicing

Mark Harman and Rob Hierons	Chris Fox	Sebastian Danicic & John Howroyd
Brunel University	King's College	Goldsmiths College
Uxbridge	University of London	University of London
Middlesex	Strand	New Cross
UB8 3PH	London WC2R 2LS	London SE14 6NW
United Kingdom	United Kingdom	United Kingdom

Keywords: Conditioned program slicing, pre- and post- conditions, program comprehension

Abstract

This paper shows how analysis of programs in terms of pre- and post- conditions can be improved using a generalisation of conditioned program slicing called pre/post conditioned slicing. Such conditions play an important role in program comprehension, reuse, verification and re-engineering.

Fully automated analysis is impossible because of the inherent undecidability of pre- and post- conditions. The method presented here reformulates the problem to circumvent this. The reformulation is constructed so that programs which respect the pre- and post-conditions applied to them have empty slices. For those which do not respect the conditions, the slice contains statements which could potentially break the conditions. This separates the automatable part of the analysis from the human analysis.

1 Introduction

It is well known that program verification and related condition-based analyses can only be partially automated [6, 25, 48]. There are often several 'eureka' steps in the analysis process which any automated system is insufficiently powerful to detect. Theorem provers and validity checkers [43, 42, 4] will either answer 'verified', 'not verified', or 'unsure' when presented with propositions concerning programs. Of course, there will always be some programs and specifications for which the answer is 'unsure' due to the inherent undecidability of the questions embodied in the conditions.

This paper focuses on pre- and post-conditions as a mechanism through which a program's behaviour can be captured. Pre- and post- conditions play an important role in software maintenance and evolution. They form a basis for program comprehension [15, 47] and are useful in reuse [8], migration [9] and re-engineering [10].

The paper introduces a new analysis method called pre/post conditioned slicing[1]. This method addresses the undecidability problem by reformulating the analysis question as a program simplification problem. The basis of the idea is to use the pre- condition and the negation of the post-condition as the basis for a generalised form of conditioned slicing which combines forward and backward conditioning.

Informally[2], the pre/post conditioned slicing process is based upon the following rule:

> "Statements are removed if they cannot lead to satisfaction of the negation of the post condition, when executed in an initial state which satisfies the pre-condition."

Suppose this rule is applied to a program which correctly implements the pre- and post- condition. In this case, it will be possible to remove *all* statements from the program to form the slice according to the above rule. This observation is at the heart of the approach presented here. The essence of the idea is to use conditioned slicing to form slices according to this rule. The rule is constructed so that the slicing process will remove all statements which can be shown to preserve the behaviour of the original program with respect to the pre- and post-condition. This leaves behind just those statements which are either incorrect (with respect to the pre- and post- condition) or which are 'innocent' but

[1] This should not to be confused with the $p - slices$ introduced by Comuzzi and Hart [13] to determine the statements which merely *affect* a predicate.

[2] This is made precise by a formal unification of forward and backward conditioning presented in Section 6.

```
pre-condition:  x ≠ 0
if (x<0)
    x = x*x;
negated post-condition:  x ≤ 0
```

Figure 1. A Simple Illustrative Example

cannot be detected to be so by slicing, due to the inherent undecidability of the problem.

For example, consider the code fragment in Figure 1. Suppose the pre-condition is $x \neq 0$ and the post-condition is $x > 0$. Observe that the program respects the pre- and post-condition. That is, if the pre-condition is satisfied, then the program's execution will satisfy the post-condition.

This is revealed by the generalised conditioned slicing defined by the informal rule above, because both the assignment and its controlling predicate can be removed according to this rule.

The rest of this paper is organised as follows. Section 2 summarises existing work on backward and forward conditioning and slicing and informally introduces the generalization of conditioned slicing which combines backward and forward approaches. In section 3 this generalisation is used as the basis for the pre/post conditioned slicing analysis method. Section 4 illustrates the method with an example of the analysis of an engine controller program. Section 5 briefly describes the way in which the approach could be applied to verification, reuse and program comprehension. Section 6 provides a theoretical foundation for the method, formalizing the unification of forward and backward program conditioning introduced in Section 2. Section 7 concludes with directions for future work.

2 Unifying Forward and Backward Conditioning

Program slicing was introduced by Weiser [48], and since then has been developed as a tool for static [32] and dynamic analysis [35, 1]. Slicing has been applied to many problems in software maintenance and evolution, such as re-engineering [10, 36], testing and regression testing [5, 23, 27], decomposition, integration and modification [21, 20, 31], decompilation [12], program comprehension [15, 24, 25] and debugging [26, 37, 33].

Slices are constructed according to a criterion known as the 'slicing criterion'. Weiser's slicing criterion consisted of a set of variables of interest V and a point of interest within the original program n. Statements which cannot affect the values of variables in V at n are removed to form the slice. For example, consider the program in section (a) of Figure 2. Slicing this program with respect to the criterion $(\{\mathtt{x}\}, 8)$ yields the slice in section (b).

Conditioned slicing was introduced by Canfora, Cimitile and De Lucia [7, 14, 15]. It forms a bridge between the two extremes of static and dynamic analysis. It augments the traditional static slicing criterion with a condition which captures a set of initial program states of interest. This allows a programmer to further specialize a program by eliminating statements which do not contribute to the computation of the variables of interest when the program is executed in one of the initial states of interest.

The conditioned slicing criterion is thus a triple, (p, V, n) where p is some initial condition of interest and (V, n) are the two components of the 'static' slicing criterion. For example, the conditioned slice of the original program in section (a) of Figure 2 for the criterion $(\mathtt{x} > 0, \{\mathtt{x}\}, 8)$ is shown in section (c) of the figure. This slice is also the dynamic slice for all input sequences in which the first element, x of the sequence satisfies the condition $x > 0$.

More recently [19], the present authors introduced backward conditioning. In backward conditioning, a statement is deleted if, when executed, it cannot lead to the condition being satisfied. By contrast, in forward conditioning a statement is deleted if, when the condition is satisfied, it cannot be executed. By combining backward and forward conditioning a generalization of conditioned slicing is obtained. This generalisation makes conditions as flexible as static slicing criteria, which can also be inserted at arbitrary program points and also operate in either a forward or backward direction.

The ability to specify backward and forward conditions requires a notation which indicates the direction of conditions. This is achieved using arrow notation [19]. Using the notation, the slicing criterion for the traditional conditioned slice in section (c) of Figure 2 is reformulated as

$$\{({}_{\downarrow}\lceil \mathtt{x} > 0 \rceil, 1), (\{\mathtt{x}\}, 8)\}$$

Forward conditioning assists the programmer by considering the effect of propagating state information forward from a condition. This addresses questions of the form

> "what would happen if the program continued from here in some state satisfying p."

Backward conditioning is similar to forward conditioning; a condition is inserted into the program text. The effect however, is the 'mirror-image' of that for forward conditioning. For example, section (d) of Figure 2 shows the effect of backward conditioning on the original program in section (a) with respect to the condition $\{({}^{\uparrow}\lfloor \mathtt{x} < 0 \rfloor), 8), (\{\mathtt{x}\}, 8)\}$. This slice removes code which cannot leave the program in a final state satisfying $\mathtt{x} < 0$.

Backward conditioning assists the programmer by considering the effect of propagating state information backward from a condition. This addresses questions of the form "how could the program have arrived here in some state satisfying p."

This paper advocates an approach which combines forward and backward conditions. Forward conditions will be used to propagate forward the meaning of the pre-condition, while backward conditions will be used to propagate backward, the meaning of the post condition.

For example, consider the conditioned slicing criterion $\{(^{\uparrow}\lfloor \mathtt{x} < 0\rfloor), 8), (_{\downarrow}\lceil \mathtt{x} > 0\rceil, 1), (\{\mathtt{x}\}, 8)\}$. That is, if x starts out non-negative, then no statement can cause it to become negative.

Section 6 formalised the generalisation of conditioned slicing introduced informally in this section.

3 The Pre/Post Conditioned Slicing Analysis Method

Forward conditioning with respect to a condition p removes statements which cannot be executed in any state which satisfies p. Backward conditioning with respect to a condition q removes statements whose execution cannot lead to a state which satisfies q. In analysis of a program with respect to pre-condition α and post-condition ω, it would be expected that no execution could start in a state satisfying α and lead to a state satisfying $\neg\omega$. Therefore, if the program is conditioned with respect to the forward condition α and the backward condition $\neg\omega$ the conditioned slice *should* to be empty. That is, if the program is *correct* with respect to α and ω, the slice *should* to be empty.

This provides a new approach to the verification and analysis of the pre- and post-conditions. Rather than attempting to prove that the pre-condition implies the post-condition, an attempt is made to reduce the slice to the empty program when applying generalised conditioned slicing to the pre-condition and the negation of the post-condition.

In situations where the pre-condition implies the post-condition, slicing will be capable of reducing the slice to empty. Unfortunately, due to undecidability, empty slices may not be achievable in practice. However, in situations where the pre-condition does imply the post-condition, but *no* automated technique can establish this fact, the slicing approach improves upon true/false verification. This is because it will typically reduce the size of the problem remaining for human analysis, by slicing away some of the statements of the program which can be shown to be 'innocent'.

Of course, not all programs *are* correct with respect to their pre- and post-conditions. Indeed, for some applications, such as program comprehension, the pre- and post-condition used may be merely *speculative* rather than *required*. In situations where the pre-condition does not imply the post condition, the slicing approach is *also* more appropriate than a true/false verification approach. This is because slicing will remove those statements which can be shown to respect the pre- and post-conditions, leaving the 'suspect' statements behind.

```
float Tmp1, Tmp2;
Tmp1= CurrentVal-GoodVal;
Tmp2= GoodVal-CurrentVal;
if (Tmp1>SmoothThresh ||
    Tmp2 SmoothThresh)
  { Ct= Ct+1;
    if (Ct<Thresh)
        OutputVal= GoodVal;
        else
        { OutputVal= CurrentVal;
          GoodVal= CurrentVal;
          Ct= 0; }
  }
  else
  { OutputVal= CurrentVal;
    GoodVal= CurrentVal;
    Ct= 0; }
```

Figure 3. Engine Control System Code

4 A Worked Example

This section illustrates the pre- and post- condition slicing method using the code fragment in Figure 3. This code is a fragment of production software that forms part of an engine controller [40].

The code fragment has six parameters. Three of these (`CurrentVal`, `SmoothThresh`, and `Thresh`) are used for input only. Two parameters (`GoodVal` and `Ct`) are used for input and output while `OutputVal` is used for output only.

In the specification of the program, there are two constraints [40]:

" The value of `Ct` lies between 0 and Thresh (inclusive)."

The method can be applied to the problem of examining this invariant constraint. Where the constraint is *required*, the pre/post condition slicing method is therefore an aid to verification. Where it is merely *speculated* by a human analyst in an attempt to understand the behaviour of the code, the method is an aid to comprehension.

To examine whether or not the constraint is maintained by the program the constraint will be asserted as a pre-condition and post-condition. Using the pre/post conditioned slicing method, this will entail conditioning with respect to the following conditions:

1 scanf("%d",&x); 2 y=2*x; 3 if (y>x) 4 { x=x+1; 5 y=y*y;} else 6 { x=x*2; 7 y=y-x;} 8 printf("%d",x);	1 scanf("%d",&x); 2 y=2*x; 3 if (y>x) 4 x=x+1; else 6 x=x*2;	1 scanf("%d",&x); 2 y=2*x; 3 if (y>x) 4 x=x+1;	1 scanf("%d",&x); 2 y=2*x; 3 if (y>x) else 6 x=x*2;
(a) Original	(b) slice on $(\{\mathtt{x}\}, 8)$	(c) Conditioned on $_{\downarrow}\lceil \mathtt{x} > 0 \rceil$	(d) Conditioned on $^{\uparrow}\lfloor \mathtt{x} > 0 \rfloor$

Figure 2. Comparison of forward and backward conditioning

```
Tmp1= CurrentVal-GoodVal;
Tmp2= GoodVal-CurrentVal;
if (Tmp1>SmoothThresh ||
    Tmp2>SmoothThresh)
  Ct= Ct+1;
  else
    ;
```

Figure 4. The Result of Slicing

1. Pre-condition: `Ct>=0 && Ct<=Thresh` (forward condition)
2. Negated Post-condition: `Ct <0 || Ct >Thresh` (backward condition)

Generalised conditioned slicing with respect to this condition yields the empty slice. This establishes the fact that if the value of `Ct` is within the required range at the beginning of the procedure then no statement will take it out of this range at the end of the procedure.

Of course, it is possible that the value of `Ct` slips in and out of range during the execution of the program, although it always finishes up in range.

In order to analyse this situation, assertions an be inserted into the body[3] of the program at each assignment to `Ct`. In this case there is only one. The procedure is sliced with the condition `Ct < 0 || Ct > Thresh` immediately after the assignment to `Ct` and the pre-condition `Ct>0 && Ct<=Thresh`. Slicing on this criterion yields the fragment in Figure 4.

Implementations of conditioned slicing [14, 7] construct conditioned slices in terms of path conditions. The ConSIT implementation [14] also simplifies these path conditions using the Isabelle theorem prover [43, 42]. This can be useful in examining the conditions which could lead to an exception (such as the one captured by the value of `Ct` going out of range during a computation).

[3]The formalisation of generalised conditioning presented in Section 6 allows assertions to be located at arbitrary points in programs, though it is often convenient to think of them as pre- and post-conditions.

The path condition in this case is:

$$\begin{array}{l} Ct \geq 0 \wedge Ct \leq Thresh \wedge Ct + 1 > Thresh \wedge \\ (\mathit{CurrentVal} - \mathit{GoodVal} > \mathit{SmoothThresh} \\ \;\vee \mathit{GoodVal} - \mathit{CurrentVal} > \mathit{SmoothThresh}) \end{array}$$

Simplification produces the following conditions for an exception to be raised by the increment to `Ct`.

$$\begin{array}{l} Ct = Thresh \wedge \\ ((\mathit{CurrentVal} - \mathit{GoodVal} > \mathit{SmoothThresh}) \vee \\ \;\;(\mathit{GoodVal} - \mathit{CurrentVal} > \mathit{SmoothThresh})) \end{array}$$

In summary, the pre/post conditioned slicing method has been used to examine the constraint $0 \leq Ct \leq Thresh$. The method has shown that when execution starts in a state which satisfies the constraint, then no statement causes it to fail at the end of the code fragment. However, the method also identified a situation where the constraint may be broken *during* execution and produced a simplified exception condition under which the constraint could be broken during execution.

5 Application to Maintenance and Evolution

This section briefly considers the way in which conditioned slicing with respect to pre- and post- conditions can be applied to comprehension, reuse and verification.

5.1 Comprehension

Comprehension often starts with questions about program behaviour. These questions represent speculative conditions about the execution of a program, and may be incorrect. As there may be many questions that may be asked of an existing system, it would clearly be attractive to automate the process of answering them.

Questions about a program's behaviour can typically be coded as assertions at various points in the program code, so a pre- and post-condition approach is applicable here.

Unfortunately there are two problems with any approach to the automation of pre- and post- condition analysis for speculative questions:

1. Being merely speculative, the question asked may be false, but an automated system which simply answers 'no' to the question yields little insight into *why* the answer is 'no'.
2. Being generally undecidable, such questions may not be completely answerable by an automated system.

As stated earlier, the second of these two problems cannot be avoided, although it is circumvented in the conditioned slicing approach, by approximating the answer through program simplification; the better the simplification the closer the approximation.

The first of these two concerns is more important for program comprehension, where *many* questions will have the answer 'no'. In this situation the conditioned slicing approach yields valuable insight by returning, not just the answer 'no', but also a reduced program which contains statements which potentially cause the answer to be 'no'. This yields additional insight into why the pre- and post-conditions failed to adequately capture the behaviour of the program.

5.2 Reuse

In order to re-use program components it is often necessary to specify the pre- and post- conditions of the code required. These conditions can be used to search code in a data base of candidate reuse components. The pre- and post- condition search approach to reuse was first suggested by Katz et al. [34], Perry [45] and Rollins and Wing [46]. A survey of the approach is presented by Mili [38].

Systems which automate the search for such components typically use theorem proving to find possible matches [39, 44]. Unfortunately, in some cases there may be no exact match, but there may be several components which represent near matches. Fischer et al.[17] report retrieval rates of 0.49, which is low because of the power required of the theorem proving technology is simply too great.

The approach advocated here can be thought of as a way of finding *approximate* component library matches. Instead of using theorem proving to locate an exact match for the component sought, the system can search for the smallest slices produced (to within some chosen tolerance of 'smallness'). If a perfect match could be found by theorem proving alone, then the slicing approach will locate the same component by returning an empty slice for it. However, where theorem proving alone would be unable to locate *any* components, the slicing approach will return a set of candidates and will indicate the sections of code (in the conditioned slice) which potentially deviate from the required semantics. The size of the conditioned slice can thus be used as a measure of component fit.

5.3 Verification

Program verification often consists of examining the effect of the program in terms of assertions [47]. The axiomatic method, introduced by floyd [18] and developed by Hoare, Dijkstra and others [28, 29, 3, 16] is centred around the development of program semantics in terms of assertions. The approach advocated here provides a method for analysing the effects of these assertions, simplifying the human effort required to verify the program.

Many specifications are written in state-based languages such as statecharts or SDL [22]. Such specifications are effectively extended finite state machines: there is a finite set of logical states, transitions between these states, and an internal memory or store. Where the internal store is finite, there is an overall finite state structure and it is possible to check properties quasi-exhaustively using model checking [41, 30, 11]. However, even where the internal store is finite, the number of possible values for this may make model checking infeasible. It might be possible to use an approach similar to conditioned slicing to reduce this problem: given a property being checked, aspects that are not relevant to this property might be sliced away. This could improve the efficiency, and thus extend the applicability, of model checking.

6 Formal Foundations

This section presents a formalisation of generalised conditioning in which statement blocks are eliminated from the program if there are no execution paths through which the conditions of interest are true. These conditions are declared using `assert` statements. There may be any number of them, and they may appear at any position in the program. The program statements that are left are then those that are on at least one path where all of the required conditions are true (or more precisely, not *provably* false).

In effect there are three salient cases:

1. If a statement s is inaccessible given a preceding `assert` statement, then that statement can be eliminated (as in standard conditioning).

2. If every path through s contains an assertion which is provably false, then s can be eliminated.

3. If either of the above, then s is left in the program.

This is slightly more general than what is required. In the present setting, the focus lies in asserting two conditions, one expressing a *precondition*, at or near the beginning of the program, and a second expressing a *postcondition*—in the form of an assertion of the *negation* of a postcondition—at the end of the program. This means that the above cases can be re-expressed as follows:

1. If a statement s is inaccessible given the precondition, then that statement can be eliminated.

2. If s is accessible given the precondition, and all paths through s satisfy the postcondition, then s can be eliminated.

3. If s is accessible given the precondition, but there is at least one execution path through the statement that cannot be shown to satisfy the postcondition, then s is left in the program.

The formalisation of `assert` and of conditioning aims to preserve the symbolic semantics of the program, before and after conditioning. In case (2) above, simply deleting statement s is not guaranteed to preserve the semantics. To satisfy this requirement, "eliminating" a statement s is taken to include the replacement of s by `assert(false);` a statement which has the effect of removing from further consideration any trouble free paths which lead to the satisfaction of the postcondition when s was in place. In the program schema:

```
...
if (c) {s1}
...
s2
...
```

if case (2) applies to s_1, and case (3) applies to s_2 then this will lead to:

```
...
if (c) {assert(false); }
...
s2
...
```

preserving the fact that s_2 is only troublesome in the original program when the condition c is false, by "trapping" all paths in which c is true. If these paths were not trapped in this way, and s_2 was simply deleted, then the semantics of the program would have been changed, because there would then be new paths through s_2, namely the previously acceptable paths that used to be modified by s_1.

In case (1) above, elimination can take the form of statement deletion or replacement by `assert(false)`. According to this formalisation, a program satisfies a postcondition, with a given precondition, if all the statements have either been deleted, or replaced by `assert(false)`.

In the general case that is formalised here, if the salient condition occurs before a statement that is left in the resultant conditioned program, then this is like saying that the statement is potentially accessible on paths where the condition is true. If the salient condition follows the statement, then it can be said that the statement is on a path which potentially contributes to the satisfaction of the condition. Although this characterisation might help with intuitions, in the semantics to be presented below, there is no formal distinction between these two cases.

6.1 Symbolic Semantics

Before defining the symbolic semantics of the key statements, assignments, conditionals, while loops and input statements, it is convenient to define notions of an update to the states in a set of path, symbolic-state expressions[4]. The expression $\simeq$ to stand for partial equality, which is not defined if either argument is $\perp$. In this analysis, $\perp$ will be the state that results on execution paths that do not contribute to the truth of the required condition(s).

The notation $\Sigma \circ [x/e]$ is used to represent the result of updating the symbolic states in the set of path-states Σ by the substitution $[x/e]$, and $\Sigma \uparrow p$ is used when the paths in a set of path-states Σ are augmented by an additional path constraint p. The expression $\pi \triangleright \sigma$ is used to indicate a path-state pair; if path condition π holds, then symbolic state σ will arise. The term $\mathcal{I}_\sigma(e)$ is the symbolic evaluation of program expression e in symbolic state σ.

Definitions 1–4 give the formal infrastructure required to define the formal symbolic semantics of the core language in Definition 5. In particular, Definitions 1 and 2 are required for clauses 1–4 of Definition 5, and Definitions 3 and 4 are required for clause 5, which covers `while` loops.

The notion of a set of path-states is then generalised to include a distinguished value $\perp$ that is used in the symbolic semantics of `assert`, given in Definition 6. Note that the intentions is that `assert` should be considered a first class citizen of the programming language. It is not included in Definition 5 merely to aid the clarity of the presentation of the formal theory.

Finally, Definition 8 formalises generalised condition itself.

Definition 1 *Composition of a set of path-states Σ with a substitution $[x/y]$:*

$$\Sigma \circ [x/e] \simeq \{(\pi \triangleright \sigma') : (\pi \triangleright \sigma) \in \Sigma, \sigma' = \sigma[x/\mathcal{I}_\sigma(e)]\}$$

[4] Here the exposition is simplified by not distinguishing between conditional expressions of the programming language and those of the meta-language in which the notion of a program's symbolic semantics is formalised.

Definition 2 *Update of a set of path-states Σ with a new path condition p:*

$$\Sigma \uparrow p \simeq \{(\pi' \triangleright \sigma) : (\pi \triangleright \sigma) \in \Sigma, \pi' = \pi \cup \mathcal{I}_\sigma(p)\}$$

This is sufficient to state the semantics of assignment and conditional statements. For `while` loops the notion of a "touched" variable also needs to be defined.

Definition 3 *The touched variables $\tau(s)$ of a statement s are those which might be assigned a value within s:*

$$\tau(s) \simeq \{v : \exists(\pi \triangleright \sigma) \in ((\emptyset \triangleright \emptyset) \cdot s).\exists e.(v = e) \in \sigma\}$$

Touched variables can be safely approximated by the set of defined variables (those which occur on the left-hand side of an assignment statement [2]).

A way of "resetting," or "clearing" a set of variables V in a set of path-state pairs that might have been touched within a loop body can be defined. First it can be noted that resetting of an individual variable v can be defined by the following expression:

$$\rho_{\{v\}}(\Sigma) \simeq \{\pi \triangleright \sigma' : \pi \triangleright \sigma \in \Sigma, \sigma' = \sigma[v/\mathcal{U}(\sigma)]\}$$

where $\mathcal{U}(\sigma)$ picks out a constant symbolic value that is unique in σ. This can be generalised to give a recursive definition for the renaming of a set of variables V in a set of path-states Σ.

Definition 4 *$\rho_V(\Sigma)$ is the path condition Σ except where variables V are assigned unique constant values. Case (i) is the base case when the set of variables in the empty set:*

$$\rho_\emptyset(\Sigma) \simeq \Sigma$$

Case (ii) is the recursive case:

$$\begin{aligned}\rho_V(\Sigma) \simeq \{&\pi \triangleright \sigma' : \pi \triangleright \sigma \in \Sigma',\\ &\sigma' = \sigma[v/\mathcal{U}(\sigma)], \Sigma' = \rho_{V-\{v\}}\,\Sigma,\\ &\text{for some some } v \in V\}\end{aligned}$$

Now everything is in place to define the symbolic semantics of the usual statement types:

Definition 5 *$\Sigma \cdot s$ is used to mean the set of path-states that arise from the execution of statement s in the context of path-states Σ.*

1. $\Sigma \cdot (v{=}e) \simeq \Sigma \circ [v/e]$
2. $\Sigma \cdot \texttt{scanf("\%d", \&}v\texttt{)} \simeq \rho_{\{v\}}(\Sigma)$
3. $\Sigma \cdot (s;s') \simeq (\Sigma \cdot s) \cdot s'$
4. $\Sigma \cdot (\texttt{if}\ c\ s\ \texttt{else}\ s') \simeq (\Sigma \uparrow c) \cdot s \cup (\Sigma \uparrow \texttt{not}\ c) \cdot s'$
5. $\Sigma \cdot (\texttt{while}\, c\, s) \simeq (\Sigma \uparrow \texttt{not}\ c) \cup ((\rho_{\tau(s)}\,\Sigma) \uparrow c \cdot s) \uparrow \texttt{not}\ c$

This defines the symbolic execution semantics for a fragment of C [14].

This can be exploited in the definition of backward conditioning. Essentially, for each statement s it must be determined whether all execution paths through s lead to the negation of the required condition(s). If so, then we can remove that statement.

First, the symbolic semantics must be extended to account for the existence of statements of the form `assert(`c`)`. The notation $\Sigma^\perp$ is used to stand for a value that is either a set of path-states, or the distinguished value $\perp$, which arises when c can be shown to be false on all relevant paths.

Definition 6 *Symbolic execution of* `assert`*:*

$$\Sigma^\perp \cdot \texttt{assert}(c) = \begin{cases} \perp \text{ iff } \pi \vdash \neg\mathcal{I}_\sigma(c) \\ \quad \text{for all } (\pi \triangleright \sigma) \in \Sigma^\perp \\ \Sigma \uparrow c \text{ otherwise} \end{cases}$$

This is distinct from the previous semantics, in that it encompasses an element of evaluation.

For completeness, the relevant behaviours of $\perp$ can be given:

Definition 7 *All of $\Sigma^\perp \cdot s, \Sigma^\perp \uparrow \pi, \Sigma^\perp \circ \sigma, \rho_V(\Sigma^\perp)$ are $\perp$ if $\Sigma^\perp = \perp$, and when $a \neq \perp$, and $b \neq \perp$ then $a \simeq b \equiv a = b$.*

Various choices are available in determining the appropriate action on conditioning a program. On determining that s is irrelevant for the purposes of obtaining the desired condition, it would be possible, for example, to rewrite conditional statements of the form

1. `if` $c\ s$ `else` s'
2. `if` $c\ s'$`else` s

as simply:

$$s'$$

However, it should be noted that, in general, if this approach were adopted for the case where a conditional expression occurs prior to an `assert` statement, then execution paths that would have gone through s will now incorrectly go through s'. This changes the semantics of the transformed program. To avoid this, all paths that would have gone through the deleted statement would have to be trapped, perhaps using an `assert` statement. In the case of the above conditionals, if c is always true (false, respectively) the statement can be replaced by

1. `assert(!`c`);` s'
2. `assert(`c`);` s'

respectively.

An alternative would be to simply delete the irrelevant individual statements and replace them with `assert(false);`. For the purpose of exposition this latter version is formalised. Note that it is possible to consider strengthening this treatment to cover some special cases, for example when the `else` part of a conditional is empty.

Definition 8 $\mathcal{C}^{\Sigma}_{R}(s)$ *denotes the post-conditioned version of the program s where all statements have been elided if they cannot lead to the satisfaction of the required conditions, as expressed by* `assert` *statements.*[5] *The subscript (s initially) is used to record the program continuation, and the superscript ($\{\emptyset \triangleright \emptyset\}$ initially) is used to record the path-states of the program so far.*

1. $\mathcal{C}^{\Sigma}_{R}(v{=}e) = (v{=}e)$
2. $\mathcal{C}^{\Sigma}_{R}(\texttt{scanf\{"\%d", \&}v\}) = \texttt{scanf\{"\%d", \&}v\}$
3. $\mathcal{C}^{\Sigma}_{R}(s;s') = \mathcal{C}^{\Sigma}_{s';R}(s); \mathcal{C}^{\Sigma\cdot s}_{R}(s')$
4. $\mathcal{C}^{\Sigma}_{R}(\texttt{if}\ c\ s\ \texttt{else}\ s')$
$$= \begin{cases} \texttt{if}\ c\ \mathcal{C}^{\Sigma\uparrow c}_{R}(s)\ \texttt{else}\ \{\texttt{assert(false)}\} \\ \quad \text{when } (\Sigma^{\perp}\uparrow\neg c)\cdot(s';R) = \perp \\ \texttt{if}\ c\ \{\texttt{assert(false)}\}\ \texttt{else}\ \mathcal{C}^{\Sigma\uparrow\neg c}_{R}(s') \\ \quad \text{when } (\Sigma^{\perp}\uparrow c)\cdot(s;R) = \perp \\ \texttt{if}\ c\ \mathcal{C}^{\Sigma\uparrow c}_{R}(s)\ \texttt{else}\ \mathcal{C}^{\Sigma\uparrow\neg c}_{R}(s') \text{ otherwise} \end{cases}$$
5. $\mathcal{C}^{\Sigma}_{R}(\texttt{while}\ c\ s)$
$$= \begin{cases} \texttt{while}\ c\ \{\texttt{assert(false)}\} \\ \quad \text{when } ((\rho_{\mathcal{T}(s)}\ \Sigma^{\perp})\uparrow c\cdot s)\uparrow \texttt{not}\ c\cdot R = \perp \\ \texttt{while}\ c\ \mathcal{C}^{\Sigma\uparrow c}_{R}(s) \text{ otherwise} \end{cases}$$

The parts 4 and 5 of this definition can be strengthened so that when

$$\pi \vdash \mathcal{I}_{\sigma}(c) \text{ for all } (\pi \triangleright \sigma) \in \Sigma$$

the statement s[6] is obtained in place of $\texttt{if}\ c\ s\ \texttt{else}\ s'$ and when

$$\pi \vdash \neg\mathcal{I}_{\sigma}(c) \text{ for all } (\pi \triangleright \sigma) \in \Sigma$$

the statement s[7] and $\{\}$[8], respectively, are obtained in place of $\texttt{if}\ c\ s'\ \texttt{else}\ s$ and $\texttt{while}\ c\ \{\}$. These additional clauses give us the behaviour of the original conditioning procedure.

The net result is that:

1. statements that are only on infeasible paths are either replaced by `assert(false)`, or are removed in the case of the strengthened analysis (as in the original version of conditioning);
2. with the strengthened version, statements following a point of non-termination are removed, (as in the original version of conditioning);
3. statements on feasible paths that reach a condition, and guarantee that the condition is false are replaced by `assert(false)` (this covers the "backward" conditioning case).

[5] In this context, "elided" can mean either a well formed deletion, or some kind of statement "colouring."

[6] Or one of (i) $\texttt{if true}\ s\ \texttt{else}\ s'$, (ii) $\texttt{if}\ c\ \texttt{else}\ \{\}$. The latter matches the original implementation of conditioning.

[7] Or one of (i) $\texttt{if false}\ s'\ \texttt{else}\ s$, (ii) $\texttt{if}\ c\ \{\}\ \texttt{else}\ s$. The latter matches the original implementation of conditioning.

[8] Or one of (i) $\texttt{while false}\ s$ (ii) $\texttt{while}\ c\ \{\}$. The latter matches the original implementation of conditioning.

7 Conclusion and Future Work

This paper has introduced and formailsed the pre/post conditioned slicing method, which combines forward and backward conditioning to provide a unified framework for conditioned program slicing. It shows that the method can be used to analyse programs in terms of their pre- and post-conditions, by formulating conditioned slicing criteria from the pre-condition and the negation of the post-condition. Statements not in the slice are those which must behave correctly with respect to the pre- and post- condition, while those which remain are 'suspect' and form the basis for further analysis.

The paper illustrates the application of the pre/post conditioned slicing method with an example of an engine controller and pre- and post- conditions which specify constraints on its operation. The paper also considers the application of the method to reuse and program comprehension.

More work is required to evaluate the effectiveness of this approach as a tool for reuse, verification and comprehension. The authors plan empirical studies of the efficacy of the approach in these three areas based on the existing ConSIT conditioned slicing tool [14].

References

[1] AGRAWAL, H., AND HORGAN, J. R. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, June 1990), pp. 246–256.

[2] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, techniques and tools*. Addison Wesley, 1986.

[3] APT, K. R. Ten years of hoare's logic: A survey – part. *ACM Transactions on Programming Languages and Systems 3*, 4 (Oct. 1981), 431–483.

[4] BARRETT, C., DILL, D., AND LEVITT, J. Validity checking for combinations of theories with equality. In *Formal Methods In Computer-Aided Design* (November 1996), M. Srivas

and A. Camilleri, Eds., vol. 1166 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 187–201. Palo Alto, California, November 6–8.

[5] BINKLEY, D. W. The application of program slicing to regression testing. In *Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier, 1998, pp. 583–594.

[6] BOURDONCLE, F. Abstract debugging of higher-order imperative languages. In *SIGPLAN '93 Conference on Programming Language Design and Implementation* (1993), pp. 46–55.

[7] CANFORA, G., CIMITILE, A., AND DE LUCIA, A. Conditioned program slicing. In *Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier Science B. V., 1998, pp. 595–607.

[8] CANFORA, G., CIMITILE, A., DE LUCIA, A., AND LUCCA, G. A. D. Software salvaging based on conditions. In *International Conference on Software Maintenance (ICSM'96)* (Victoria, Canada, Sept. 1994), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 424–433.

[9] CANFORA, G., CIMITILE, A., LUCIA, A. D., AND LUCCA, G. A. D. Decomposing legacy programs: A first step towards migrating to client–server platforms. In 6^{th} *IEEE International Workshop on Program Comprehension* (Ischia, Italy, June 1998), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 136–144.

[10] CANFORA, G., LUCIA, A. D., AND MUNRO, M. An integrated environment for reuse reengineering C code. *Journal of Systems and Software 42* (1998), 153–164.

[11] CHAN, W., ANDERSON, R. J., BEAME, P., BURNS, S., MODUGNO, F., AND NOTKIN, D. Model checking large software specifications. *IEEE Transactions on Software Engineering 24* (1998), 498–519.

[12] CIFUENTES, C., AND FRABOULET, A. Intraprocedural static slicing of binary executables. In *IEEE International Conference on Software Maintenance (ICSM'97)* (1997), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 188–195.

[13] COMUZZI, J. J., AND HART, J. M. Program Slicing using Weakest Preconditions. In *FME '96: Industrial Benefit and Advances in Formal Methods* (Oxford, Mar. 1996), J. W. M.-C. Gaudel, Ed., vol. 1051 of *LNCS*, Springer Verlag, pp. 557–575.

[14] DANICIC, S., FOX, C., HARMAN, M., AND HIERONS, R. M. ConSIT: A conditioned program slicer. In *IEEE International Conference on Software Maintenance (ICSM'00)* (San Jose, California, USA, Oct. 2000), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 216–226.

[15] DE LUCIA, A., FASOLINO, A. R., AND MUNRO, M. Understanding function behaviours through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension* (Berlin, Germany, Mar. 1996), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 9–18.

[16] DIJKSTRA, E. W. *A discipline of programming*. Prentice Hall, 1972.

[17] FISCHER, B., KIEVERNAGEL, M., AND SNELTING, G. Deduction-based software component retrieval. In *IJCAI95 Workshop on formal Approaches to the Reuse of Plans, Proofs, and Programs* (1995).

[18] FLOYD, R. W. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, J. T. Schwartz, Ed., vol. 19 of *Symposia in Applied Mathematics*. American Mathematical Society, Providence, RI, 1967, pp. 19–32.

[19] FOX, C., HARMAN, M., HIERONS, R. M., AND DANICIC, S. Backward conditioning: a new program specialisation technique and its application to program comprehension. In 9^{th} *IEEE International Workshop on Program Comprehesion (IWPC'01)* (Toronto, Canada, May 2001), IEEE Computer Society Press, Los Alamitos, California, USA. To appear.

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

[21] GALLAGHER, K. B., AND LYLE, J. R. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering 17*, 8 (Aug. 1991), 751–761.

[22] HAREL, D., AND POLITI, M. *Modeling reactive systems with statecharts: the STATEMATE approach*. McGraw-Hill, 1998.

[23] HARMAN, M., AND DANICIC, S. Using program slicing to simplify testing. *Software Testing, Verification and Reliability 5*, 3 (Sept. 1995), 143–162.

[24] HARMAN, M., AND DANICIC, S. Amorphous program slicing. In 5^{th} *IEEE International Workshop on Program Comprehesion (IWPC'97)* (Dearborn, Michigan, USA, May 1997), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 70–79.

[25] HARMAN, M., FOX, C., HIERONS, R. M., BINKLEY, D., AND DANICIC, S. Program simplification as a means of approximating undecidable propositions. In 7^{th} *IEEE International Workshop on Program Comprehesion (IWPC'99)* (Pittsburgh, Pennsylvania, USA, May 1999), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 208–217.

[26] HARMAN, M., SIVAGURUNATHAN, Y., AND DANICIC, S. Analysis of dynamic memory access using amorphous slicing. In *IEEE International Conference on Software Maintenance (ICSM'98)* (Bethesda, Maryland, USA, Nov. 1998), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 336–345.

[27] HIERONS, R. M., HARMAN, M., AND DANICIC, S. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability 9*, 4 (1999), 233–262.

[28] HOARE, C. A. R. An Axiomatic Basis of Computer Programming. *Communications of the ACM 12* (1969), 576–580.

[29] HOARE, C. A. R., AND WIRTH, N. An axiomatic definition of the programming language Pascal. *Acta Informatica 2*, 4 (Dec. 1973), 335–355.

[30] HOLZMANN, G. J. The model checker SPIN. *IEEE Transactions on Software Engineering 23* (1997), 279–295.

[31] HORWITZ, S., PRINS, J., AND REPS, T. Integrating non–interfering versions of programs. *ACM Transactions on Programming Languages and Systems 11*, 3 (July 1989), 345–387.

[32] HORWITZ, S., REPS, T., AND BINKLEY, D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems 12*, 1 (1990), 26–61.

[33] KAMKAR, M. *Interprocedural dynamic slicing with applications to debugging and testing*. PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden, 1993. Available as Linköping Studies in Science and Technology, Dissertations, Number 297.

[34] KATZ, S., RICHTER, C. A., AND THE, K.-S. PARIS: A system for reusing partially interpreted schemas. In *Proceedings of the 9th International Conference on Software Engineering* (1987), IEEE Computer Society Press, pp. 377–385.

[35] KOREL, B., AND LASKI, J. Dynamic program slicing. *Information Processing Letters 29*, 3 (Oct. 1988), 155–163.

[36] LAKHOTIA, A., AND DEPREZ, J.-C. Restructuring programs by tucking statements into functions. In *Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier, 1998, pp. 677–689.

[37] LYLE, J. R., AND WEISER, M. Automatic program bug location by program slicing. In 2^{nd} *International Conference on Computers and Applications* (Peking, 1987), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 877–882.

[38] MILI, A., MILI, R., AND MITTERMEIR, R. T. A survey of software reuse libraries. *Annals of Software Engineering 5* (1998), 349–414. Systematic Software Reuse.

[39] MILI, R., MILI, A., AND MITTERMEIR, R. T. Storing and retrieving software components: A refinement based system. *IEEE Transactions on Software Engineering 23*, 7 (July 1997), 445–460.

[40] NIGEL TRACEY, JOHN CLARK, K. M., AND MCDERMID, J. Automated test-data generation for exception conditions. 61–79.

[41] OZDEMIR, K., AND URAL, H. Protocol validation by simultaneous reachability analysis. *Computer Communications 20* (1997), 772–788.

[42] PAULSON, L. C. Isabelle: A generic theorem prover. *Lecture Notes in Computer Science 828* (1994), xvii + 321.

[43] PAULSON, L. C. Strategic principles in the design of Isabelle. In *CADE-15 Workshop on Strategies in Automated Deduction* (Lindau, Germany, 1998), pp. 11–17.

[44] PENIX, J., BARAONA, P., AND ALEXANDER, P. Classification and retrieval of reusable components using semantic features. In *Proceedings: 10th Knowledge-Based Software Engineering Conference* (1995), IEEE Computer Society Press, pp. 131–138.

[45] PERRY, D. E. The Inscape Environment. In *Proceedings: 11th International Conference on Software Engineering* (1989), IEEE Computer Society Press / ACM Press, pp. 2–11.

[46] ROLLINS, E. J., AND WING, J. M. Specifications as search keys for software libraries. In *Proceedings of the 8th International Conference on Logic Programming* (June 1991), K. Furukawa, Ed., MIT Press, pp. 173–187.

[47] ROSENBLUM, D. S. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering 21*, 1 (Jan. 1995), 19–31.

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