
CONsIT: A Fully Automated Conditioned Program Slicer



Chris Fox^{1,*}, Sebastian Danicic², Mark Harman³ and Robert M. Hierons³

¹ *Department of Computer Science, University of Essex, Wivenhoe Park, Colchester CO4 3SQ, U.K.
E-mail: foxcj@essex.ac.uk*

² *Department of Computing, Goldsmiths College, University of London, New Cross, London SE14 6NW,
U.K. E-mail: mas01sd@gold.ac.uk*

³ *Department of Information Systems and Computing, Brunel University, Uxbridge, Middlesex, UB8 3PH,
U.K. E-mail: mark.harman@brunel.ac.uk and rob.hierons@brunel.ac.uk*

SUMMARY

Conditioned slicing is a source code extraction technique. The extraction is performed with respect to a slicing criterion which contains a set of variables and conditions of interest. Conditioned slicing removes the parts of the original program which cannot affect the variables at the point of interest, when the conditions are satisfied. This produces a conditioned slice, which preserves the behaviour of the original with respect to the slicing criterion.

Conditioned slicing has applications in source code comprehension, reuse, restructuring and testing. Unfortunately, implementation is not straightforward because the full exploitation of conditions requires the combination of symbolic execution, theorem proving and traditional static slicing. Hitherto, this difficulty has hindered development of fully automated conditioning slicing tools.

This paper describes the first fully automated conditioned slicing system, CONsIT, detailing the theory that underlies it, its architecture and the way it combines symbolic execution, theorem proving and slicing technologies. The use of CONsIT is illustrated with respect to the applications of testing and comprehension.

KEY WORDS: Conditioned Slicing, program conditioning, slicing, symbolic execution, path analysis

INTRODUCTION

Program slicing is an automatic program extraction technique in which statements which cannot affect a slicing criterion are removed to form a slice. In initial work on slicing [47, 48], the criterion was a program point and set of variables. For this criterion the slice is a (possibly) reduced program which maintains the effect of the original on the value of all variables at the program point of interest.

This form of slicing has come to be known as static slicing, because the slicing criterion contains no information about how the program is to be executed. In dynamic slicing [37, 2], by contrast, the static

*Correspondence to: Dr. Chris Fox <foxcj@essex.ac.uk>, Department of Computer Science, University of Essex, Wivenhoe Park, Colchester CO4 3SQ, United Kingdom

criterion is augmented with the input to the program. Parts of the program which are deleted can have no effect upon the variable of interest at the program point of interest when the program is executed on this particular input.

Whereas a static slice preserves the behaviour of the program for all inputs, a dynamic slice only preserves behaviour for a particular input. Dynamic slices therefore tend to be far smaller than static slices, but also far more specific. Dynamic slicing is most useful when the program has been executed, for example in debugging applications.

For applications where the program execution remains unspecified, a ‘more static’ approach is clearly required. However, it is often not helpful to make the criterion ‘completely static’. Often there is a desire to put some constraints on the program’s possible executions. These constraints are the ‘conditions’ of conditioned slicing. Since the condition could simply be either ‘true’ or a conjunction of equalities which define the input, it is possible (in theory at least) for conditioned slicing to subsume both static and dynamic forms of slice [7].

For example, suppose the programmer wants to understand the behaviour of the original program when some condition is satisfied, here conditioned slicing will remove parts of the program which cannot have an effect when the condition is met. Another possibility is that the program has some conditions already available (perhaps in the form of subdomain from partition analysis, from the specification, or from safety constraints). Here too, conditioned slicing can be helpful, by focusing attention on those parts of the original program which are relevant to the conditions under consideration. Two applications of conditioned slicing are considered in more detail below (see page 20).

To illustrate the difference between static, dynamic and conditioned forms of slicing, consider the program fragment in the left-most column of Figure 1. This fragment determines the type of a triangle based upon the three side lengths a , b and c . The dynamic slice for the variable r at the end of the program and for the input which sets a , b and c to 1 is shown in the central column of the figure. The static slice on the final value of r is the entire program, as every line affects the final value of r in some way. Notice that the dynamic slice is very specific. It only applies when the variables are all set to one. This criterion can be implemented by conditioned slicing, using the condition $a=b=c=1$, but a more general condition would produce the same slice, namely: $a=b=c$.

The Conditioned slice for the final value of r when the condition is

$$a = b \vee a = c \vee b = c$$

is shown in the rightmost column of the figure. As a simple illustration of the way conditioned slicing aids understanding, consider the programmer who mistakenly believes that under this condition the program stores *isosceles* in r . As the programmer expects, slicing removes the assignment of the value *scalene*, but leaves the assignment of *equilateral*, indicated the misunderstanding. If the programmer augments the condition to

$$a = b \vee a = c \vee b = c \wedge \neg(a = b \wedge b = c)$$

Then the assignment to *equilateral* disappears. Using the approach described in this paper, these conditions are inserted as `assert` statements directly into the program source. Thus the programmer can easily see the effect of various ‘assumptions’ about the program, by ‘asserting and slicing’.

Canfora et al. [7] provide a survey of work on conditioned slicing. Korel and Rilling [38] provide a survey of dynamic slicing. More general surveys of slicing can be found in papers by Tip [45], Binkley and Gallagher [4] and De Lucia [18].

<pre> if (a==b) if (a==c) r="equilateral"; else r="isosceles"; else if (a==c) r="isosceles"; else if (b==c) r="isosceles"; else r="scalene"; </pre>	<pre> if (a==b) if (a==c) r="equilateral"; </pre>	<pre> if (a==b) if (a==c) r="equilateral"; else r="isosceles"; else if (a==c) r="isosceles"; else if (b==c) r="isosceles"; </pre>
Original	Dynamic slice for $a=b=c=1$	Conditioned slice for $a = b \vee a = c \vee b = c$

Figure 1. Comparing Static, Dynamic and Conditioned Slicing

This paper describes a system called CONSIT. CONSIT implements conditioned slicing for a subset of the C programming language. Conditioned slicing involves the combination of conditioning and static slicing. Static slicing is already a well understood problem and has been the subject of several successful algorithms [48, 34, 16] and implementations[†]. Conditioning is a less well developed technology. The problem is to identify and remove paths which become infeasible when the constraints of the conditioned slicing criterion are applied. Although the identification of all such paths is not decidable, the detection of any such paths and possible reduction in slice size that such detection permits, is valuable in all slicing's applications. This is because, like other forms of slicing, conditioned slicing only needs to be a safe approximation to a (generally non computable) minimal slice.

The rest of this paper is organised as follows: first, the definition of conditioned program slicing is formalised (page 4); a brief guide to using the system is given (page 5); a running example is presented that will be used to illustrate various points (page 5); an overview of CONSIT's architecture is given (page 8); the two main components of CONSIT, the program conditioner and slicer, are then presented (page 10 and page 20, respectively); two applications are discussed (page 20); future and related work (pages 27 and 25, respectively) are followed by the conclusions (page 29).

[†]See, for example, the "Unravel Project," by James R. Lyle, Dolores R. Wallace, James R. Graham, Keith B. Gallagher, Joseph P. Poole and David W. Binkley (hissa.ncsl.nist.gov/~jimmy/unravel.html), and the "Wisconsin Program Slicing Project," by Susan Horwitz and Thomas Reps (www.cs.wisc.edu/wpis/html/).

CONDITIONED PROGRAM SLICING

The previous section provided an informal definition of conditioned slice. This section formalises this definition and draws a distinction between two forms of conditioned slice, which are termed ‘Control Dependence Sensitive’ (CDS) and ‘Control Dependence Insensitive’ (CDI) conditioning in this paper.

Definition 1 (Conditioned Slice) *A Conditioned slice is constructed with respect to a tuple, (V, n, π) , where V is a set of variables, n is a point in the program (typically a node of the Control Flow Graph) and π is some condition. A statement may be removed from a program p to form a slice, s of p , iff it cannot affect the value of any variable in V when the next statement to be executed is at point n and the initial state is such that π is satisfied at point n .*

Sometimes, it is helpful to think of two components to the definition of a conditioned slice, the definition of a conditioned program and the definition of a static slice. Conditioned slicing can then be achieved using a combination of conditioning and (static) slicing.

The definition below captures the properties of a static slice.

Definition 2 (Static Slice) *A Static slice is constructed with respect to a tuple, (V, n) , where V is a set of variables and n is a point in the program (typically a node of the Control Flow Graph). A statement may be removed from a program p to form a static slice, s of p , iff it cannot affect the value of any variable in V when the next statement to be executed is at point n .*

There are two forms of conditioning, CDS Conditioning and CDI Conditioning. The difference is captured in the definitions below, and is illustrated by the discussion which follows.

Definition 3 (Control Dependence Sensitive (CDS) Conditioned Program) *A conditioned program is constructed with respect to a condition, π . A statement may be removed from a program p to form a conditioned program, s of p , iff it cannot be executed when the initial state satisfies π .*

In CDS conditioning, a statement is only removed if it can be deduced that it will be not executed, given the constraint imposed by the initial condition, π .

Definition 4 (Control Dependence Insensitive (CDI) Conditioned Program) *A conditioned program is constructed with respect to a condition, π . A statement may be removed from a program p to form a conditioned program, s of p , iff when s and p are executed in an initial state that satisfies π , the values of all variables at all points in s agree with their values at corresponding points in p .*

In initial work on conditioning[19, 7], the CDS version of conditioning was adopted. For example, conditioning the program fragment

```

if (a>b)
  x=1;
else
  x=2;

```

with respect to the initial condition $a>b$, using CDS conditioning produces the conditioned program

```

if (a>b)
  x=1;

```

rather than the simpler

```
x=1;
```

The second form of conditioned slice is smaller and satisfies the requirements of conditioned program (that it behaves the same as the original when the conditions are met). Hereinafter, the original form of conditioned slice, based on CDS conditioning, shall be called an ‘CDS conditioned slice’, whereas the form in which predicates are removed by conditioning shall be called an ‘CDI conditioned slice’.

For example, in Figure 1 the conditioned slice in the rightmost column of the figure was both CDS and CDI, and so it is simply referred to as a conditioned slice. The dynamic slice in the central column, is also an CDS conditioned slice on the criterion $a=b=c$. However, the CDI conditioned slice for this criterion would have been:

```
r=equilateral;
```

USING CONSIT

The input to the tool is a program together with statements that give the conditioning and slicing criteria. In the current implementation of the system, these additional statements, or *annotations* are added manually. This is illustrated in the running example that is described in the next section. The original program is as given in Figure 3. The program that the system analyses includes the additional statements `assert(blind && widow && age < 50)`; and `slice = tax`; that appear in the revised program given in Figure 4. The first statement gives the conditioning criteria (see ‘Asserts’ on page 15), and the second gives the static slicing criteria (see ‘Slicing the tax program’ on page 20).

The nature of the slicing criteria requires some explanation. To simplify the implementation, the slicer always slices on the distinguished variable `slice` at the end of the program. More general slicing criteria can be stated by making assignments to `slice` at the required point(s) in the program.

In addition to a command line tool, a Web interface has also been developed, as illustrated in Figure 2. The user pastes the program code into the left text pane, adding the required conditioning and slicing annotations, and the results of conditioned program slicing appear in the right text pane.

A RUNNING EXAMPLE

For illustrative purposes, the discussion of the CONSIT system will make reference to a running example, given in Figure 3. This program codifies part of the UK tax system for the year 1999–2000.

The conditioning and slicing criteria are given as annotations to this program. In the example depicted in Figure 4, the annotations are highlighted as follows:

annotations

They state a condition which captures computations for blind widows under 50 years old. The other part of the criterion is that the variable of interest is `tax`, as computed at the end of the program. The

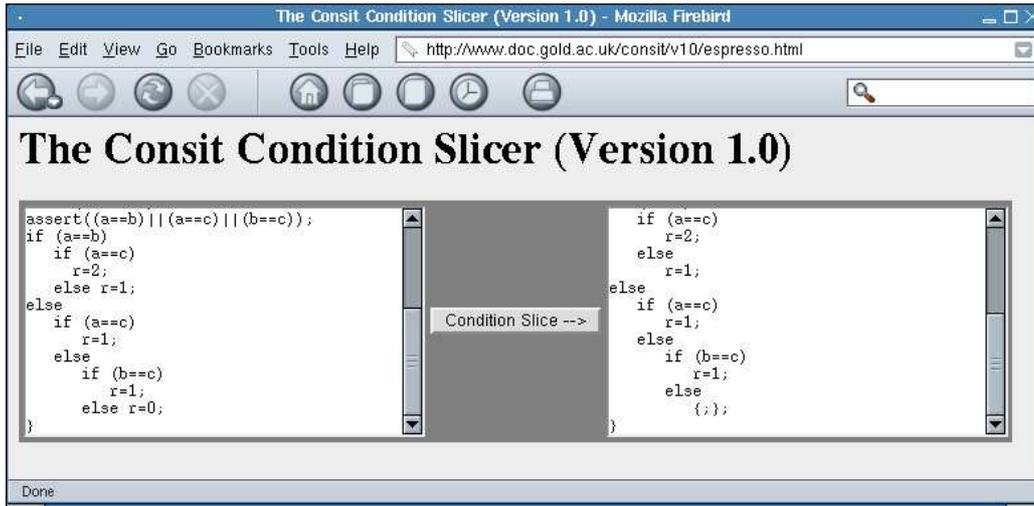
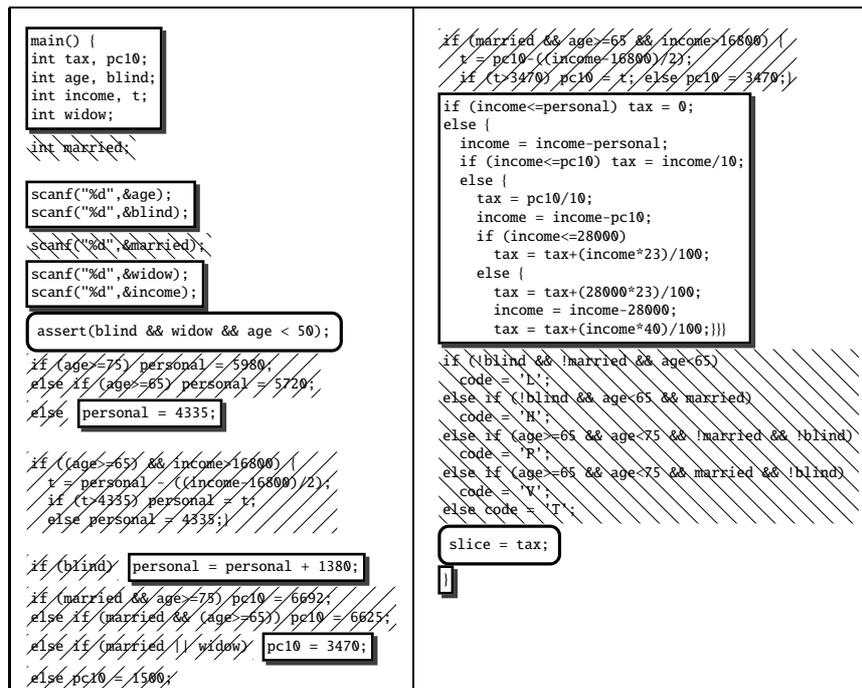


Figure 2. Web interface to ConSIT. The figure illustrates conditioning of a 'triangle test' program, with the condition that all sides of the triangle (a,b,c) are equal, focusing on the part of the code that is effected.

<pre>main() { int tax, pc10; int age, blind; int income, t; int widow; int married; scanf("%d",&age); scanf("%d",&blind); scanf("%d",&married); scanf("%d",&widow); scanf("%d",&income); if (age>=75) personal = 5980; else if (age>=65) personal = 5720; else personal = 4335; if ((age>=65) && income>16800) { t = personal - ((income-16800)/2); if (t>4335) personal = t; else personal = 4335;} if (blind) personal = personal + 1380; if (married && age>=75) pc10 = 6692; else if (married && (age>=65)) pc10 = 6625; else if (married widow) pc10 = 3470; else pc10 = 1500;</pre>	<pre>if (married && age>=65 && income>16800) { t = pc10-((income-16800)/2); if (t>3470) pc10 = t; else pc10 = 3470;} if (income<=personal) tax = 0; else { income = income-personal; if (income<=pc10) tax = income/10; else { tax = pc10/10; income = income-pc10; if (income<=28000) tax = tax+(income*23)/100; else { tax = tax+(28000*23)/100; income = income-28000; tax = tax+(income*40)/100;}} if (!blind && !married && age<65) code = 'L'; else if (!blind && age<65 && married) code = 'H'; else if (age>=65 && age<75 && !married && !blind) code = 'P'; else if (age>=65 && age<75 && married && !blind) code = 'V'; else code = 'T'; }</pre>
--	--

Figure 3. The Taxation Program



criteria sliced code elided/inaccessible code remaining code

Figure 4. Tax for Blind Widow Under 50

annotations are part of the program which forms the input to CONSIT. In this way, the CONSIT approach is to encode the slicing criterion in the program itself.

On processing this program (with respect to the slicing criterion), the slice will be indicated using:

~~sliced code~~

to show which code is not relevant to the computation of `tax`. Code that does not lie on any feasible path (and the predicates of conditionals statements that are always true) given the conditioning criterion, are indicated as:

~~elided/inaccessible code~~

This effectively leaves just the code highlighted as

remaining code

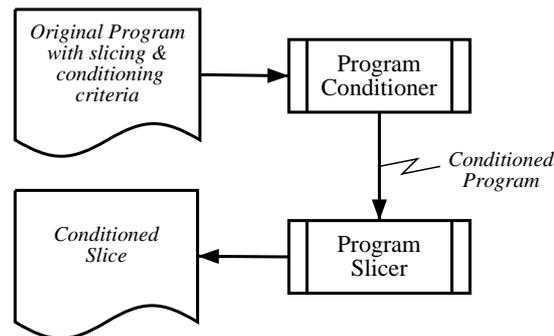


Figure 5. Top-level Architecture

which the conditioner has found to be relevant under the given conditions, namely that the computation of interest is that for blind widows under 50 years old.[‡]

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

ARCHITECTURE OF THE OVERALL SYSTEM

The system was designed and implemented as a command-line system in a LINUX environment. The overall architecture is presented in Figure 5. Conceptually the CONSIT system has two major components consisting of

1. The Conditioner
2. The Slicer

These two components are described in more detail later (see pages 10 and 20, respectively). The original program is first annotated with the slicing and conditioning criteria of interest using an `assert` statement (see ‘Asserts,’ page 15) for the conditioning criteria, and an assignment to the variable `slice` for the static slicing criteria (see ‘Slicing the tax program,’ page 20), and is then presented to the program conditioner, to produce a conditioned program, followed by the program slicer, to give the final conditioned slice.

[‡]To simplify the presentation, in the case where slicing and conditioning both have the effect of ‘removing’ a statement, the statement is marked as having just been sliced away.

Integration

The language C^b is merely a subset of C, without gotos, pointers, functions, arrays or structures, and with just the basic type of integers. *Haste* is a purpose built language with a simplified C-like syntax which was adopted to simplify the implementation of the conditioner. As they have a similar statement level abstract syntax, translation between these languages is relatively straightforward. In Appendix A and Appendix B we give the syntax of *Haste* and C^b , respectively. C^b is used as the “surface syntax” of the system, as seen by the user, so that the system as a whole produces conditioned slices for a subset of C.

Filters `c2Haste` and `Haste2C`, which translate between the object languages C^b and *Haste*, were written in Java. To pass object programs between the slicer, `sliceC` and the conditioner, `hastecond`, they first have to be filtered through the appropriate translator. CONSIT is implemented as a pipeline script which takes a C^b program and first conditions it and then slices it, as expressed by the following UNIX pipeline:

```
c2Haste | hastecond | Haste2C | sliceC
```

The lexical analysers and parsers for *Haste* and C^b —used by the language filters and the slicer—were written in Java, using the Java Compiler Tools `Jlex` and `Javacup`.

A condition slice server, `Igor` has been implemented. This simply listens at a port, waiting to receive the source text of a program that is to be condition sliced. In order to produce a conditioned slice, the server calls an appropriate composition of `sliceC` and `hastecond` and transmits the resulting output object program on the same port. Users communicate with the `Igor` server via an applet running in a Web browser (see page 5). The system can produce various output for debugging purposes.

Architecture of the Conditioner

The conditioning stage consists of (i) a symbolic executor and (ii) a condition verifier, which is used to determine which parts of a program are accessible in a given context. Both the symbolic executor, and the “wrapper” for the condition verifier, are written in ANSI Prolog. They are compiled to a single, stand-alone, executable using `SWI-Prolog`. The overall architecture of the conditioner, together with the wrappers that convert from and to C^b (the `c2Haste` pre-processor and `Haste2C` post-processor), is as depicted in Figure 6.

The condition verifier (Figure 7), calls from Prolog to the SVC “`check_valid`” program [3, 40] are made via the UNIX shell. The overhead in spawning a new process each time the SVC is invoked is considered to be insignificant relative to the time that the SVC takes to run for all but the most trivial examples. The quick start-up time for `check_valid` means that there was little need for adopting the client-server approach that was required with a previous version of CONSIT [14].

Architecture of the Slicer

The slicer is based on the Parallel Slicing Algorithm [16] which in turn is based on Mark Weiser’s original slicing algorithm [47] and is implemented in Java using Espresso, a multi-threaded slicer generator [15]. Having parsed the object program p , a representation of its Control Flow Graph (CFG) is obtained from a representation of p ’s abstract syntax. From this CFG a network of concurrent process

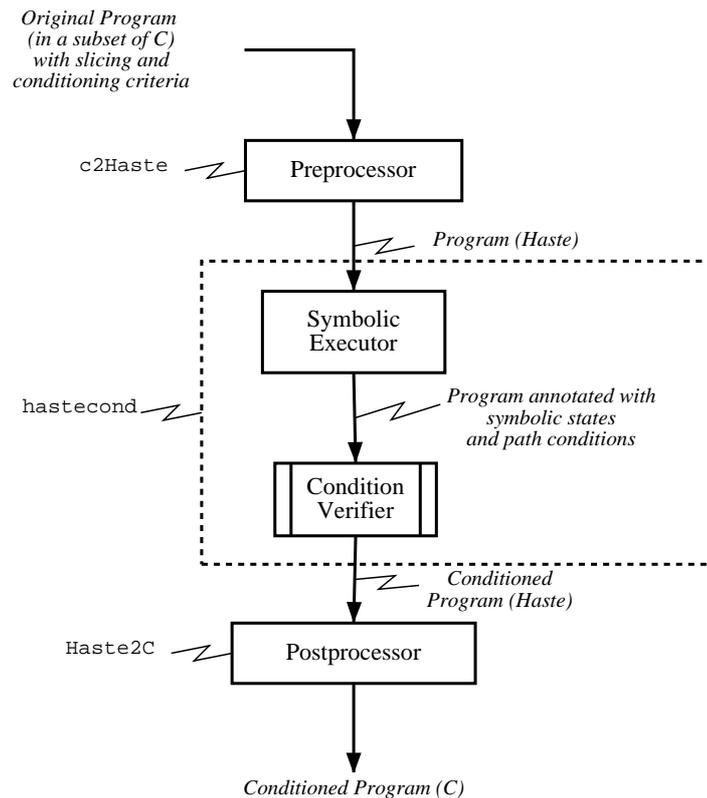


Figure 6. Architecture of the Conditioner

is compiled. Execution of this ‘slice network’ produces the set of nodes S of the CFG to be included in the slice. The resulting program slice is then reconstructed using the original abstract syntax of p together with the slice set S . Since *Haste* and C^b share the same abstract syntax, pretty printers which map from abstract to either concrete syntax are easily implemented.

The slicer supports a client-server architecture, so that the slicing software can reside on a high-performance machine (Figure 8).

THE CONDITIONER SUBSYSTEM

Conceptually, the conditioner component consists of two stages. The first stage is *symbolic execution*. For each statement in the program, the system determines all of the paths that could have been taken

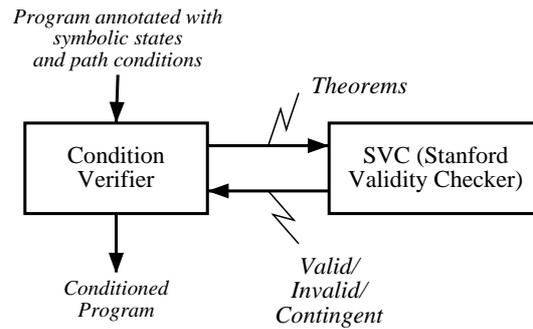


Figure 7. SVC-based Condition Verifier

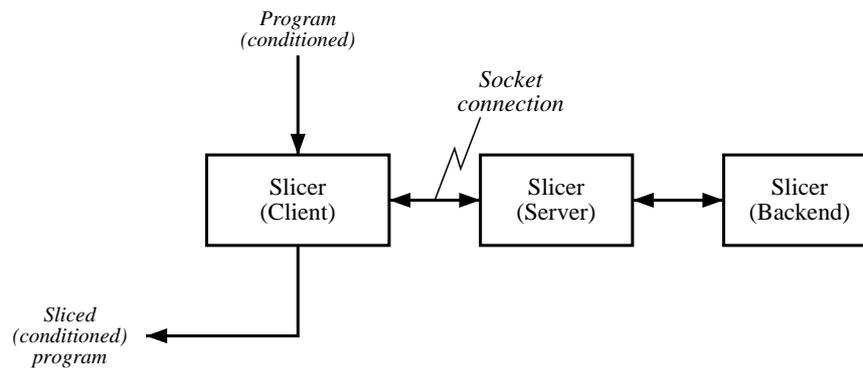


Figure 8. Slicer Architecture

to reach that statement, expressing these paths as pairs of path conditions and symbolic states. This is similar to the approach adopted in the partially automated system reported by Canfora *et al.* [7].

The symbolic execution proceeds purely syntactically; it does not, by itself, recognise or prune inconsistent paths and states. This is part of the role of the second *infeasible path analysis* stage of the conditioner.

The path analysis stage proceeds by checking whether each statement is accessible by determining if there is at least one consistent path condition. If a statement is inaccessible, then it can be replaced by `skip`.

Symbolic Execution

For each possible path to a statement, the system determines the conditions under which the path is taken, together with a representation of the symbolic state of the variables on that path. As the particular values of the variables is unknown, the path conditions and states are represented symbolically, in the form of inequalities and expressions in a first order formalism, where all variables within a particular path-state pair are implicitly universally quantified. Each path-state pair is equivalent to an expression of the form:

$$\langle path\ condition \rangle \wedge \langle state \rangle$$

There will be several paths (p_1, \dots, p_n) and corresponding states (s_1, \dots, s_n) to each statement. These states, and path conditions, are those that apply *before* the execution of the statement. The complete symbolic state for a statement can be thought of as a proposition of the form:

$$\forall \vec{v}. ((p_1 \wedge s_1) \vee (p_2 \wedge s_2) \vee \dots \vee (p_n \wedge s_n))$$

where \vec{v} is the vector of all program variables.

The concrete syntax for the set of path-states se is

$$\begin{aligned} se &::= [(p_1, s_1), \dots, (p_n, s_n)] \\ s &::= [(v_1, e_1), \dots, (v_n, e_n)] \\ p &::= [c, \dots, c] \\ c &::= e = e \mid e < e \mid \sim c \mid c \ \vee \ c \mid c \ \wedge \ c \\ e &::= \langle integer \rangle \mid e + e \mid e - e \mid e * e \mid e / e \end{aligned}$$

The symbolic executer is written as Definite Clause Grammar (DCG) in Prolog. The path-state sets for each statement in the program are produced by semantic annotations on the DCG rules.

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 essence, the set of paths and symbolic states that can arise are determined as described below (for each case).

Assignments

$(\langle variable \rangle = \langle expression \rangle)$ The new set of path-state pairs is formed by adding to each symbolic state the fact that the variable 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). Symbolic values consist of arithmetic expressions with terms consisting of constant, and symbolic constant values.

Inputs

To each symbolic state is added the fact that the variable is bound to a unique symbolic constant.

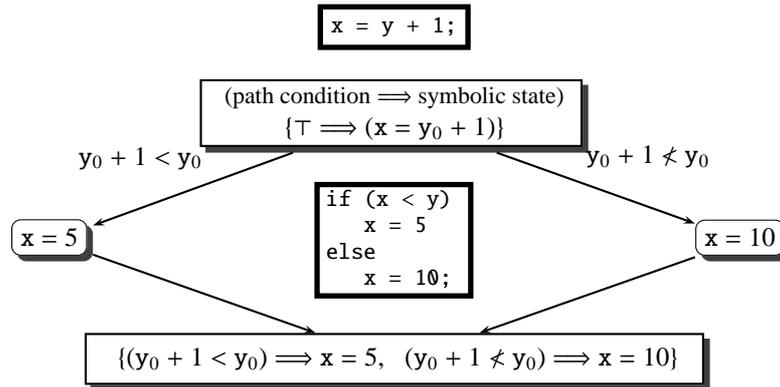


Figure 9. Symbolic Execution of a Conditional

Conditionals

(if $\langle condition \rangle s_1$ else s_2) For each path-state leading up to a conditional two path-states are created, one in which $\langle condition \rangle$ is true, the other in which it is false. The conditions are added in a symbolic form, where all variables are replaced by their symbolic values in the corresponding state. Next, s_1 is symbolically executed within the context of the path-state pairs in which $\langle condition \rangle$ has been asserted, and s_2 with the path-state pairs in which the condition is negated. The union of the results of both these symbolic executions is then formed. This thus doubles the number of path-state pairs.

Figure 9 illustrates the symbolic execution of a simple two statement program with a conditional, giving the symbolic state after executing the initial assignment, and the two possible paths through the conditional (one where the condition is true, the other in which it is false). The final symbolic state then incorporates both possibilities.

Figure 10 illustrates how combining conditionals generates the permutations of symbolic states and their associated path conditions.

Loops

(while $\langle condition \rangle s$) There are two cases to consider with loops:

1. The condition is false: the loop body s is not executed. To cover this case, negation of the condition is added to a copy of each of the current path-state pairs, replacing all variables by their symbolic values in the corresponding states, as with conditionals.
2. The condition is true: the loop body s is executed at least once. If the loop terminates, then at the final execution of s 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:

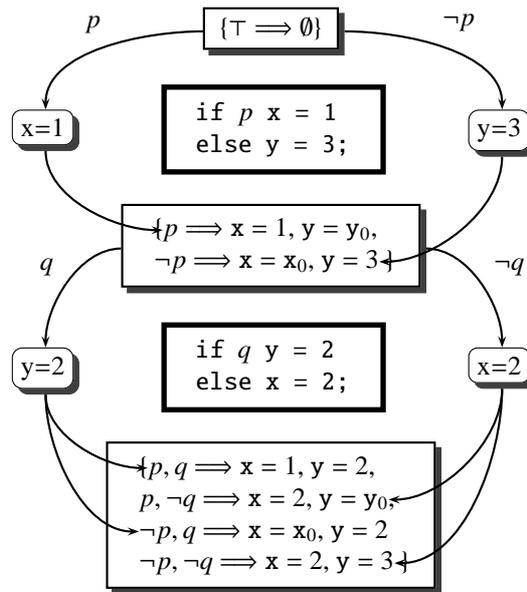


Figure 10. Combining Symbolic States

- (a) To a copy of all of the initial path-state pairs is added the fact that the condition is initially true.
- (b) Conceptually, s is then symbolically executed just once, in the context of the path-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.
- (c) As the loop condition must have been true at the start of the final iteration, and false following the final iteration, to each of the path-state pairs that result from the final iteration, the algorithm adds the loop condition, as evaluated in the symbolic states at the beginning of the final iteration, and the negation of the loop condition as evaluated at the end of the final iteration.

Of course, the loop might not terminate, but in this case the path-states that result would be inconsistent. This eventually can be ascertained during the subsequent conditioning of the program. It is not of concern during the purely syntactic construction of the set of path-states by way of symbolic execution.

The union of the path-states that result from both these eventualities is then formed. Figure 11 illustrates a basic example of this analysis, using a two-statement program containing a `while` loop. It gives the symbolic state after the execution of the initial assignment, and the two possible paths to the

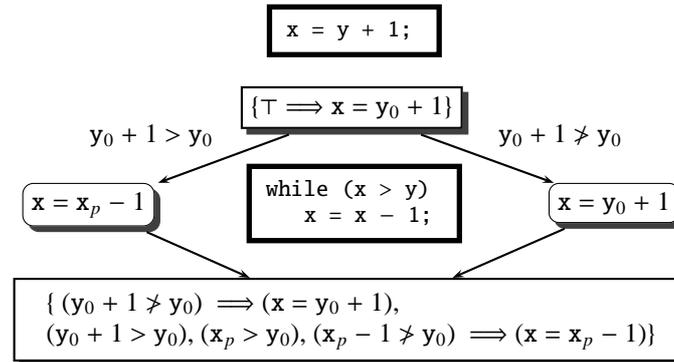


Figure 11. Symbolic Execution of a Loop

point following the `while` loop, one where the loop condition is initially `false`, the other where it is initially `true`, but becomes `false` after a number of executions of the loop. The final symbolic state incorporates both these possibilities. The case when the loop fails to terminate arises when both path conditions are `false`.

This can be thought of as implementing a form of loop *unravelling*. It is well-known that a `while` loop of the form:

```
while <condition> s
```

can be unrolled into an infinite conditional statement:

```
if <condition> { s; if <condition> { s; ... } }
```

which has a fixed point, if the loop terminates. Conceptually, here the loop is being unravelled from the other end of this potentially infinite conditional statement; in this case, the execution of `s` that is of interest is the final one, where there may have been one (or more) previous executions of `s`:[§]

```
if <condition> { ... { ... s } ... }
```

Asserts

Another special kind of statement, `assert`, is added to the language. This simply adds a condition to all of the paths through that statement. Constraints on input values can be expressed by following an `input`

[§]Although not implemented in the version of CONSTIT described in this paper, this analysis can be extended in various ways. In the context of conditioning, one interesting option is to combine loop unravelling with loop unrolling:

```
if <condition> { s; if <condition> { s; ... s } ... }
```

If the initial execution of `s` always makes the condition false, then it can be determined that the loop is equivalent to just `s` by itself.

statement with an `assert` which imposes some constraint on the value of the input variable. This means that universal constraints on input values can be imposed without having to explicitly consider the representation of universal quantification. The semantics of

```
assert <condition>
```

can be thought of as being equivalent to

```
while not <condition> skip
```

In effect, any path in which the `<condition>` does not hold will be inaccessible. Any statements which can be determined to be inaccessible when the condition is `true` will be removed (or elided) in the conditioned program.

Infeasible Path Elimination

For any statement in a program, it is possible to find the set of path-state pairs:

$$\{\langle p_1, s_1 \rangle, \dots, \langle p_n, s_n \rangle\}$$

As already noted, this can be interpreted as a proposition, implicitly within the context of existential quantification over the values of the variables \vec{v} in the program (or to be more precise, existential quantification over all possible values for the skolem constants that represent the initial and input values of the variables in the symbolic execution), giving an accessibility condition for a statement. Although state information is required in symbolic execution, it is irrelevant for the purposes of determining accessibility, so the accessibility condition of a statement that lies on paths $p_1 \dots p_n$ can be given as:

$$\exists \vec{v}. (p_1 \vee p_2 \vee \dots \vee p_n)$$

If the above proposition is not true, then the statement is inaccessible.

The basic conditioning used in the current implementation seeks to determine the accessibility of each statement in the program. Any statement that is inaccessible is replaced by `skip`.[¶]

Conditionals

Figure 12 illustrates the conditioning of the program of Figure 9. In this example, the condition is always `false`, so that both the conditional itself and the resultant symbolic states can be simplified. The conditional can be replaced by just `x = 10;`, although `CONSIT` seeks to keep the structure of the program intact by replacing the `then` statement by `skip`.

The application of this conditioning can be seen in Figure 4. For example, in the extract:

[¶]Other options are to: (1) replace booleans (or components of booleans) by `true` or `false` if they are valid or invalid respectively on all possible execution paths; or (2) replace conditionals by either the 'then' or 'else' statement in cases where the condition is always `true` or `false`, respectively. This latter approach can be extended to the conditioning of loops; if we determine that the loop condition is always `true` when first met, and `false` after one iteration, then we could replace the loop by its body.

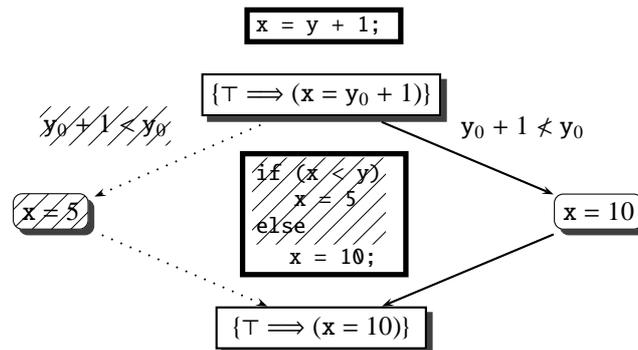


Figure 12. Conditioned Conditional (infeasible paths and statements highlighted)

```

assert(blind && widow && age < 50);
if (age >= 75) personal = 5980;
else if (age >= 65) personal = 5720;
else personal = 4335;

```

it can be seen that under the additional conditions imposed on all the paths by the `assert`, only the statement `personal = 4335` lies on a feasible path. The program can be simplified to make this clear, either by removing the inaccessible code, leaving just

```
personal = 4335;
```

rewriting the conditions,

```

if false personal = 5980;
else if false personal = 5720;
else personal = 4335;

```

or, as in the current CONSIT, by replacing the statements on infeasible paths by skip:

```

if (age >= 75) {};
else if (age >= 65) {};
else personal = 4335;

```

so preserving the structure of the original program.

Loops

Figure 13 illustrates the conditioning of the program of Figure 11. Although in this case the loop itself has not been simplified, some information has been gained that can be used when performing path

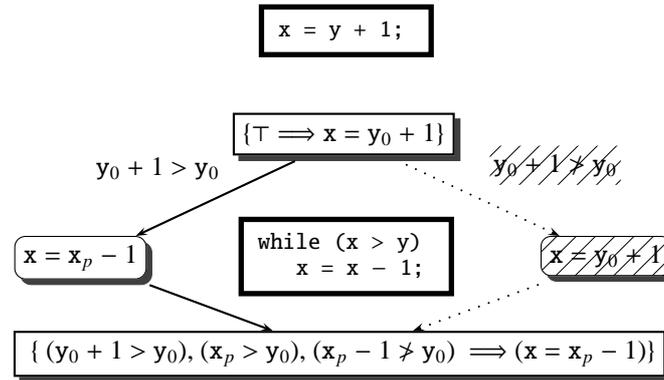


Figure 13. Conditioned Loop (infeasible paths highlighted)

analysis on statements which follow the loop. In particular, it is possible to determine that the loop will be executed at least once, and that it will terminate.

If the statement $p=5;$ were added within the loop body, and the loop was then followed by a conditional $\text{if } (p==5) s,$ then the system can determine that the statement s would be executed. Although a programmer might not put a statement of the form $p=5;$ within the loop body, such a statement might arise as a result of conditioning the loop body.

In the example given, the system determines that the final value of x is less than or equal to the initial value of y , and that $x + 1$ (i.e. the penultimate value of x) is greater than the initial value of y . This helps simplify any condition involving x and y that follows the loop. In the example given, if x and y are integers, then it can be shown that $x=y$ when the loop terminates.

Although only a partial analysis of loops, it appears to be more powerful than many symbolic execution strategies [13].

The question remains as to how we can mechanically determine whether or not a given statement is accessible on the basis of the various path-state pairs. For this we require some automatic theorem proving.

Theorem Proving

Theorem proving is a complex activity, and automatic theorem proving typically has weaknesses. However, the theorems of relevance in ConSIT typically involve inequalities over arithmetic expressions. There are specialised tools and techniques that are more appropriate for these kinds of theorems than general purpose theorem provers. One such tool is the Stanford Validity Checker (SVC) [3, 40].

SVC applies a reduction strategy together with term rewriting in order to verify expressions in first order logic with linear arithmetic inequalities. ConSIT treats SVC as a *black box*, merely translating the expressions obtained by symbolic execution into a form that is acceptable for SVC. The subset of

$$\begin{aligned}
I([(p_1, s_1), \dots, (p_n, s_n)]) &= (\text{and } (\Rightarrow (\text{not } I(p_1)) I(s_1)) \\
&\quad \dots \\
&\quad (\Rightarrow (\text{not } I(p_n)) I(s_n))) \\
I([(v_1, e_1), \dots, (v_n, e_n)]) &= (\text{and } (= v_1 I(e_1)) \\
&\quad \dots \\
&\quad (= v_n I(e_n))) \\
I([c_1, \dots, c_n]) &= (\text{and } I(c_1) \dots I(c_n)) \\
I(e_1 = e_2) &= (= I(e_1) I(e_2)) \\
I(e_1 < e_2) &= (< I(e_1) I(e_2)) \\
I(\sim c) &= (\text{not } I(c)) \\
I(c_1 \vee c_2) &= (\text{or } I(c_1) I(c_2)) \\
I(c_1 \wedge c_2) &= (\text{and } I(c_1) I(c_2)) \\
I(\langle \text{integer} \rangle) &= \langle \text{integer} \rangle \\
I(e_1 + e_2) &= \{+ I(e_1) I(e_2)\} \\
I(e_1 - e_2) &= \{+ I(e_1) \{^* I(e_2) -1\}\} \\
I(e_1 * e_2) &= \{^* I(e_1) I(e_2)\} \\
I(e_1 / e_2) &= \{^* I(e_1) \{^{**} I(e_2) -1\}\}
\end{aligned}$$

Figure 14. Translation of Symbolic Path Expressions to SVC Expressions

the concrete SVC syntax for propositions p used by the system is as follows

$$\begin{aligned}
p &::= (= e e) \mid (< e e) \mid (\text{not } p) \mid (\Rightarrow p p) \mid (\text{and } p \dots p) \mid (\text{or } p \dots p) \\
e &::= \langle \text{integer} \rangle \mid \{+ e e\} \mid \{^* e e\} \mid \{^{**} e e\}
\end{aligned}$$

The representations produced by the symbolic execution are translated as described in Figure 14.

Complexity

Experience with CONSIT suggests that the most time consuming aspect of conditioning is the validity checking of the symbolic semantics. This in turn depends upon the number of terms in the proposition to be checked.

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

The number of atomic propositions within each path is the sum of number of assignment statements a on that path and the number of atomic propositions p within the boolean component of each conditional statement and loop $O(a + p(c + l))$. So the size of the symbolic expression at the final statement of a program is $O(2^{c+l}(a + p(c + l)))$, or $O(2^n(a + pn)) = O(n2^n)$, where $n = c + l$.

In conditioning, the symbolic semantics is evaluated at each statement, not just at the end of the program. Considering just loops and conditionals, this gives $O(2^0 + 2^1 + 2^2 + \dots + 2^n) = O(n2^n)$. So the

complexity of the conditioning process is determined by the complexity of checking the accessibility of the final statement, which is determined by the number of loops and conditional expressions. Experience with `CONSIT` suggests that this complexity is manageable for unit level testing.

Programs of less than 100 lines typically take a few minutes to condition on dual symmetric processor (SMP) 500Mhz Intel Zeons. One worst case example, based on the code of Figure 4, took around one hour to condition. This was optimised to less than three minutes if the program was sliced prior to conditioning (see page 27).

THE SLICER SUBSYSTEM

The slicer is produced using the slicer-generator, Espresso [15]. Espresso is a Java program which takes a subject program p (in either C^b or *Haste*), and produces a representation of the subject program's CFG. From the CFG, Espresso produces a multithreaded slicer especially tailored for the program being sliced.

Slicing the Tax Program

Consider, for example, the tax calculation program given in Figure 4. The user gives a slicing criterion to Espresso by inserting assignments to a special variable called `slice`. The variables referenced in this assignment are taken to be the slice set. Many such assignments are allowed within the same program. This is how Espresso achieves simultaneous slicing [48].

In the tax example, there is only a single assignment to the variable `slice` representing the fact that the user wishes to perform a conventional end-slice with respect to the variable `tax`. The slice of the tax example produced by Espresso is that portion of code which has not been marked as ~~sliced away~~ in Figure 4.

APPLICATIONS OF CONDITIONED SLICING

This section describes the application of the `CONSIT` system to program comprehension and testing (using partition analysis).

Using `CONSIT` to Assist with Comprehension

Conditioned slicing can be used to assist program comprehension [19], by allowing the programmer to focus upon particular conditions and sets of variables of interest. Because the conditioned slicing process removes parts of the program which cannot affect the variables of interest when the conditions are met, the effort required to understand the behaviour of the program is reduced.

De Lucia et al. use the example in Figure 15 to illustrate this. The program calculates sums and products over positive and negative numbers. Suppose the programmer is only interested in the effect of the program on positive numbers. The `assert` statement after the `scanf` statement, has been added to the program to investigate the behaviour under this condition. The slice statement `slice = possum;` is added at the end of the program to indicate that only parts of the program which affect the variable

possum are of interest. The conditioned slice produced by CONSIT is shown in Figure 16. This is the slice suggested by De Lucia et al. [19], who constructed the slice using a semi automated system which uses a human to check the theorems that arise during the conditioning phase [17].

The slice produced by CONSIT is smaller than the corresponding static slice, because it only has to preserve behaviour for positive input values, yet it is more general than any dynamic slice, because the input is not specified; the slice is valid for an infinite set of possible input sequences. Using CONSIT, the programmer can also produce conditioned slices for the final value of `possum` when all inputs are negative. Similarly, conditioned slices can be produced for the other variables under a set of similar conditions. In this way, the programmer can build up an understanding of the program through case analysis^{||}.

Using CONSIT to Assist with Partition Analysis

In partition analysis the tester partitions the input domain D of the operation I being tested into a finite set $P = \{D_1, \dots, D_n\}$ of subsets. This partition is generated on the basis that, according to the specification, the behaviour of the system should be uniform on each D_i . Tests are then generated within the body of each subdomain and around the boundaries of the subdomains [11, 43, 49].

Partition analysis is one of the most widely used forms of specification-based (black-box) testing. However, the value of the test produced is reduced if I is not uniform on the subdomains.

Consider some subdomain $D_i \in P$, which induces a condition C_{D_i} on the input domain that is true if and only if the input is contained in D_i . Suppose that we now produce the conditioned slice S_i of I with respect to C_{D_i} . Then if I is uniform on C_{D_i} this conditioned slice is likely to be relatively simple.

Suppose that S_i is simple. The tester may either analyse S_i further, to determine whether it is correct, or use this as an indication that the behaviour is likely to be uniform in D_i . Suppose, for example, a partition has been developed for the tax program in which one of the subdomains is $C^1 \equiv \text{blind} \wedge \text{married} \wedge \text{age} \geq 75 \wedge \text{income} > 37185$. Slicing on `tax`, with condition C^1 , CONSIT produces the slice shown in Figure 17.

Here the program is simple: it contains only one path. This suggests that the behaviour is, indeed, uniform on C^1 . The tester might thus feel justified in choosing only a small number of tests that satisfy C^1 .

Suppose S_i is not simple. The tester might use this as an indication that I is unlikely to be uniform on D_i and thus that testing should be directed towards D_i . Alternatively, the tester might analyse S_i further. In particular, CONSIT can produce path conditions for the paths in S_i : the condition C_{D_i} may be refined by these conditions.

Suppose, for example, the partition developed for the tax program contains a subdomain defined by $C^2 \equiv \text{age} < 65 \wedge \neg \text{blind} \wedge \text{married} \wedge \text{income} > 4335$. Using C^2 and slicing on `tax`, CONSIT produces the conditioned slice in Figure 18.

This slice is more complex than that produced using C^1 . In particular, it contains conditional statements. This suggests that the behaviour of I is not uniform on C^2 . In this case, the path conditions,

^{||}The authors' initial application of CONSIT to the example from De Lucia et al. [19] revealed a very minor bug in the program: `possum` is miss-spelt at one line. This produced an unexpectedly small slice. While this fault could have been picked up by a compiler, it illustrated the way in which CONSIT can be used to check comprehension and reveal mistaken assumptions.

```
main() {
    int a, test0, n, i;
    int posprod, negprod, possum, negsum;
    int sum, prod;
    scanf("%d", &test0);
    scanf("%d", &n);
    i = 1;
    posprod = 1;
    negprod = 1;
    possum = 0;
    negsum = 0;
    while (i <= n) {
        scanf("%d", &a);
        assert(a>0);
        if (a > 0) {
            possum = possum + a;
            posprod = posprod * a; }
        else if (a < 0) {
            negsum = negsum - a;
            negprod = negprod * (-a); }
        else if (test0) {
            if (possum >= negsum)
                possum = 0;
            else negsum = 0;
            if (posprod >= negprod)
                posprod = 1;
            else negprod = 1; }
        i=i+1; }
    if (possum >= negsum)
        sum = possum;
    else sum = negsum;
    if (posprod >= negprod)
        prod = posprod;
    else prod = negprod;
    slice=possum;
}
```

Figure 15. Comprehension Example

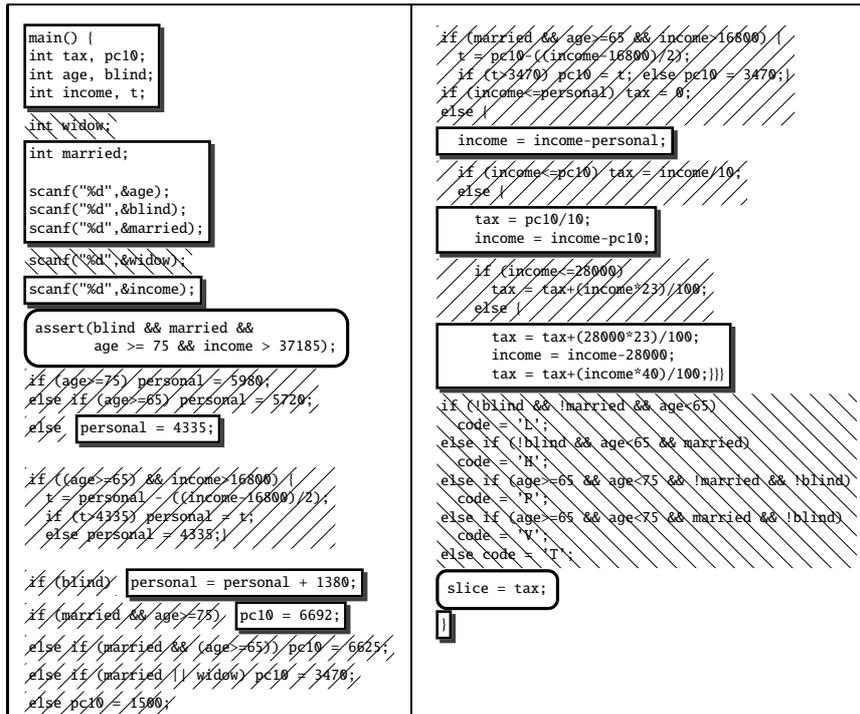
```

main() {
  int a, test0, n, i;
  int posprod, negprod, possum, negsum;
  int sum, prod;
  scanf("%d", &test0);
  scanf("%d", &n);
  i = 1;
  posprod = 1;
  negprod = 1;
  possum = 0;
  negsum = 0;
  while (i <= n) {
    scanf("%d", &a);
    assert(a>0);
    if (a > 0) {
      possum = possum + a;
      posprod = posprod * a;
    }
    else if (a < 0) {
      negsum = negsum - a;
      negprod = negprod * (-a);
    }
    else if (test0) {
      if (possum >= negsum)
        possum = 0;
      else negsum = 0;
      if (posprod >= negprod)
        posprod = 1;
      else negprod = 1;
    }
    i=i+1;
  }
  if (possum >= negsum)
    sum = possum;
  else sum = negsum;
  if (posprod >= negprod)
    prod = posprod;
  else prod = negprod;
  slice=possum;
}

```

criteria sliced code elided/inaccessible code remaining code

Figure 16. Conditioned Slice of the Comprehension Example (Figure 15) with respect to $a>0$ and $possum$



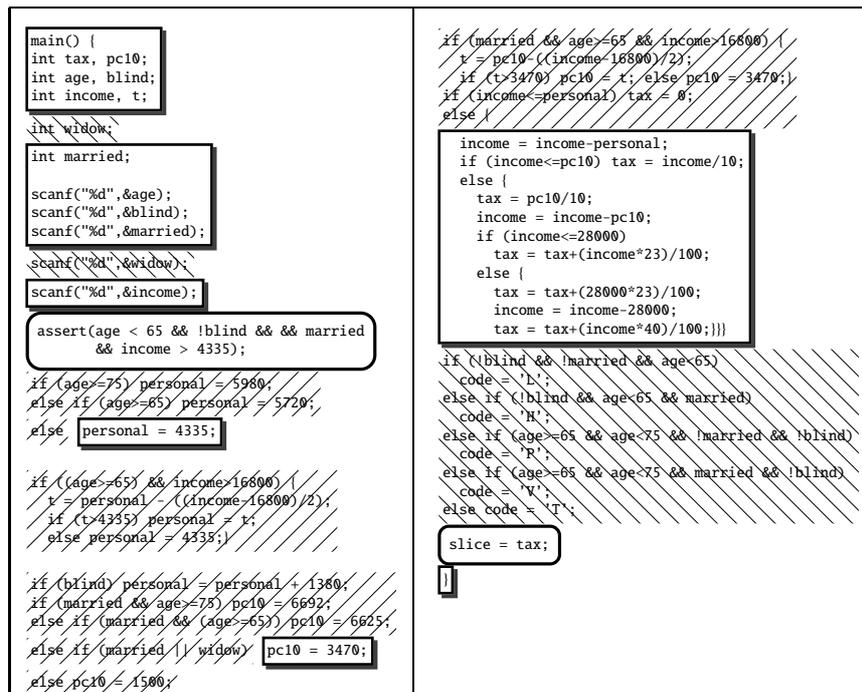
criteria sliced code elided/inaccessible code remaining code

Figure 17. Conditioned Slice of the tax example (Figure 4) with respect to C^1 and tax

contained in the slice, may be determined using CONSIT. These may be used to refine C^2 to give the conditions: $C^2 \wedge income \leq 7805$; $C^2 \wedge income > 7805 \wedge income \leq 35805$; and $C^2 \wedge income > 35805$. Tests based on the refined conditions may now be used.

An interesting observation relates to the possibility of the implementation containing special cases that do not exist in the specification. Such faults are traditionally extremely difficult to find using black-box test techniques, since there is no information in the specification to direct testing towards them. Here, however, CONSIT allows the tester to identify these extra special cases and produce tests for them.

The application of CONSIT to partition testing is considered in more detail elsewhere [32].



criteria sliced code elided/inaccessible code remaining code

Figure 18. Conditioned slice of the Tax example (Figure 4) with respect to C^2 and tax

RELATED WORK

Program slicing originated with Mark Weiser's 1979 doctoral thesis [47]. Weiser's formulation of slicing [47, 48] was a static one, captured by the slicing criterion, which was a set of variables and a program point. Weiser's original algorithm for static slicing was based upon the iterative solution of control and data flow equations. A more efficient algorithm was suggested by Ottenstein and Ottenstein [41]. This suggestion led to the development of the System Dependence Graph, introduced by Horwitz, Reps and Binkley [34].

In 1988, Korel and Laski introduced dynamic slicing [37], representing the first move away from static forms of slicing and the consideration of different forms of slicing criteria. The dynamic slice is constructed for a specific input and so it is typically thinner than its static counterpart. However, it is less general than the corresponding static slice. Since debugging typically takes place after a program has been executed, dynamic slicing is particularly useful in debugging [1, 36].

Conditioned slicing was introduced by Canfora et al. [8] in 1994. Field, Ramalingam and Tip [22] introduced a similar technique called constrained slicing, which also uses conditions to specialize program. In constrained slicing, parts of the program which cannot contribute to the values of variables of interest are identified. These program parts can be at the subexpression level and are identified as ‘holes’ in the syntax presented to the user. As such, a constrained slice is not necessarily an executable subprogram, as a conditioned slice is. As introduced by Field et al., constrained slices are also constructed in a different way to conditioned slices, using a term rewriting system to perform both slicing and conditioning on an intermediate representation, rather than as slicing and conditioning phases.

Korel and Rilling [38] survey dynamic slicing techniques and applications. Canfora, Cimitile and De Lucia [7] survey conditioned slicing and its relationship to other slicing methods. Other surveys of slicing can be found in the work of Tip [45], Binkley and Gallagher [4] and De Lucia [18]. An introductory overview is provided by Harman and Hierons [31].

Work on conditioned slicing parallels developments in partial evaluation (also known as mixed computation). Partial evaluation grew out of the work of Futamura [27] and Ershov [21]. Partial evaluation improves program performance by pre-computing or ‘partially evaluating’ (at compile time) computations which can be determined to be static. Often the opportunities for such partial evaluation are improved by providing partial information about the execution of a program. This leads to a program, called the residual program, which is specialised with respect to the partial input information. This specialization mirrors the specialization created by program slicing. The difference is that, in partial evaluation, specialization is produced by executing static computations, whereas in slicing, specialization is achieved by removing all computations which are not relevant to the slicing criterion.

Reps and Turnidge [42] described the similarities and differences between partial evaluation and static slicing. There are also close parallels between generalised forms of partial evaluation and conditioned slicing. In 1990, Venkatesh, suggested a form of slicing which provides 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 therefore subsumes static slicing (because the input prefix can be empty) and subsumes dynamic slicing (because the prefix can be an entire input sequence). Quasi static slicing was motivated by work on partial evaluation. The concept of quasi static slicing is subsumed by conditioned slicing, which is the most general form of slicing. In 1987, Futamura and Nogi [28], introduced a generalization of partial evaluation, which generalizes the way in which partial evaluation specializes a program with respect to partial input information. Using general partial evaluation, the input information is captured by a predicate, just as the conditioned slicing criterion uses a predicate to capture initial state information.

Other authors have considered the possibility of exploiting the presence of conditions in programs. This work was initiated by the seminal work of Turing [46], Floyd [25] and Hoare [33], who introduced the idea of programming with assertions. The work aims to exploit the presence of conditions to prove properties of interest statically [6] and dynamically [24, 35, 44], to extract components [23] and to remove infeasible paths before the consideration of control-flow based text coverage criteria [29]. The work aims to exploit information about execution represented by the assertions in order to analyse the programs, but does not seek to alter the programs with respect to this information. Typically, these assertions are geared towards capturing the semantics of the program, rather than extracting special cases. By contrast, slicing attempts to remove parts of the program which are irrelevant to

the assertions, and so it is expected that the assertions will capture only a projection of the complete program semantics.

Symbolic execution itself has been applied to the problem of software specialisation. The problem of dealing with loop invariants has also been addressed in this context [12].

Work has also been conducted on the generation of assertions both statically [5] and dynamically [20]. The CONSIT system assumes that assertions are present. For conditioned slicing these typically arise in program understanding re-engineering and reuse activities [8, 10, 9, 19], however, work on static and dynamic generation of assertions provides another possible source.

CONSIT concerns 'forward' conditioning, because information about states arising from conditions is propagated in a 'forward' direction to remove parts of the program which cannot be executed when the conditions are met. This contrasts with backward conditioning [26], in which information is propagated back from conditions to identify those parts of the program whose execution cannot lead to a state which satisfies the condition of interest. Lee *et al.* also consider a form of mixed forward conditioning and backward slicing, where the conditions are pre conditions and the backward slices are constructed for variables mentioned in post conditions [39]. This extracts a program component similar to pre/post conditioned slicing [30], which consists of forward and backward conditioning on pre and post conditions respectively. However, the pre/post conditioned slice is more precise than the slices considered by Lee *et al.* because it is construed with respect to the possible outcomes of the condition's evaluation, rather than merely the variables upon which this outcome depends. Future work will consider the extension of CONSIT to handle both forward and backward conditioning.

FUTURE WORK

This section considers three topics for the future development of the CONSIT system.

Optimisations

Given the O complexity of the conditioning process, it is important to seek to reduce the complexity of the analysis where possible. One way in which this can be achieved is to "fold" the conditioning and symbolic execution processes together: as described, all possible paths to each statement are generated, and it is necessary to check the accessibility of each one. An obvious improvement would be to exploit the monotonicity of the propositions that have to be analysed: if a path becomes infeasible, then it will remain infeasible for all subsequent statements. Such paths can be "pruned" once their inaccessibility has been determined. In some circumstances, this will have a significant effect on the size of the propositions handed to the theorem prover.

Another optimisation concerns the ordering of the conditioning and slicing processes in the larger system. Both slicing and conditioning effectively remove statements from a program. Slicing is not as complex as conditioning. Although in some cases the ordering of slicing and conditioning can effect the optimality of the final result, usually it does not. In fact, it can be shown that the result of slicing followed by conditioning, followed in turn by slicing is equivalent to conditioning followed by slicing. Experience with CONSIT suggests that if a program is sliced before it is conditioned, then there is a dramatic improvement in overall performance because the size of the program to be conditioned is typically shorter than the original. Even if the size of the program is reduced only slightly, the

improved performance of the conditioning step makes it worth the effort of having to slice the program a second time. In one case, a variant of the tax example (Figure 4) that initially took around an hour to execute on dual symmetric multiprocessor (SMP) 500MHz Intel Xenons when conditioning before slicing (page 19), took less than three minutes when slicing was performed before conditioning.

Because the execution time of the system depends on the nature of the source code that is being analysed, it is hard to state the typical size of a program that the system is able to handle in terms of the number of lines of code. The authors have found that the unoptimised system typically copes with units of fewer than a hundred lines in minutes. Larger units can take in the order of hours to analyse. By slicing before conditioning, CONSIT can handle units of several hundred lines in a few minutes if the slicing steps leaves fewer than a hundred lines. Initial experiments with path pruning suggest that a properly optimised system should reliably be able to handle units of several hundred lines in length in a few minutes on current hardware.

Currently, only the slicer can make full use of the parallelism of an SMP (Symmetric Multi-Processor) machine. It should be possible for the conditioner to make some use of an SMP environment by invoking multiple copies of the SVC simultaneously.

Statement Rewriting

As described, the conditioner effectively merely deletes inaccessible statements. Other simplifications are possible. For example, in the conditional expression

```
if (c) s1 else s2
```

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

In a similar vein, with the loop

```
while (c) s
```

if it can be shown that c is always true before the first execution of the loop, and false after the first execution of s , then the entire loop can be replaced by s .

Non-linearities

Nonlinearities arise with inequalities containing terms of the form ab where neither a nor b can be shown to be constant. SVC cannot handle non-linear inequalities in their full generality, which means that some inaccessible paths may be assumed to be accessible. This is safe, but suboptimal. In some cases, simplifications are possible which eliminate the nonlinearity.

However, not all such simplifications are exploited within CONSIT, and there does not appear to be a fully decidable procedure for their simplification. Pending a more rigorous analysis, an acceptable workaround is to replace nonlinear terms by unevaluated functions. The system will then still be able to determine that a statement is inaccessible provided that its inaccessibility is not contingent upon the evaluation of nonlinear terms.

To aid the comprehension of the output of the conditioner, it is possible to indicate which statements have lead to the introduction of a nonlinearity, and which statements' putative accessibility is contingent upon the status of a nonlinearity in the symbolic semantics.

CONCLUSIONS

This paper has introduced the theory, design, implementation and applications of the ConSIT system for conditioned program slicing. To the authors knowledge this is the first fully automated conditioned slicing system.

The system combines theorem proving, symbolic execution and slicing, each of which are useful tools in their own right. These component tools are integrated using a combination of a pipe/filter and a client-server architecture. The theorem proving ability is achieved using SVC. Slicing is achieved using the Espresso multi-threaded Java slicing tool and symbolic execution is achieved using a purpose-built symbolic executor written in Prolog.

The application of the tool to program comprehension and software testing is illustrated with two simple examples.

APPENDIX A. Syntax of *Haste*

Haste is the language used by the conditioning subsystem (see page 9). Its syntax is as follows.

```

<program> ::= <block> .
           | <Sequence> .
<block>   ::= { <sequence> }
<sequence> ::= <statement>
           | <statement> <sequence>
<statement> ::= SKIP
           | <var> := <expression>
           | INPUT <var>
           | IF <condition> <block>
           | ELSE <block>
           | WHILE <condition> <block>
           | COND <condition>
<expression> ::= <variable>
           | <integer>
           | - <expression>
           | ( <expression> + <expression> )
           | ( <expression> - <expression> )
           | ( <expression> * <expression> )
<condition> ::= <expression> = <expression>
           | <expression> < <expression>
           | NOT <condition>
           | ( <condition> OR <condition> )
           | ( <condition> AND <condition> )

```

APPENDIX B. SYNTAX OF C^b

C^b is the subset of C that is understood by ConSIT (see page 9). Its syntax is as follows.

```

<program>      ::= main() {<declist> <statementlist>}
<declist>     ::= <empty>
                | <declist> <decl>
<decl>        ::= <type> <varlist>;
<varlist>     ::= <IDENTIFIER>
                | <varlist>, <IDENTIFIER>
<type>        ::= int | char
<statement>   ::= ;
                | { <statementlist> }
                | <IDENTIFIER> = <expression>;
                | scanf(<STRING>, <ampersandlist>);
                | if (<expression>) <statement>
                | if (<expression>) <statement> else <statement>
                | while (<expression>) <statement>
                | assert (<expression>);
<statementlist> ::= <empty>
                | <statementlist> <statement>
<ampersandlist> ::= &<IDENTIFIER>
                | <ampersandlist>, &<IDENTIFIER>
<expression>  ::= <unop> <expression>
                | <expression> <binop> <expression>
                | <IDENTIFIER>
                | <INTEGERCONST>
                | <CHARCONST>
                | (<expression>)

```

REFERENCES

1. Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 23(6):589–616, June 1993.
2. Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, New York, June 1990.
3. Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. Validity checking for combinations of theories with equality. In *FMCAD'96*, November 1996.
4. David Wendell Binkley and Keith Brian Gallagher. Program slicing. In Marvin Zelkowitz, editor, *Advances of Computing, Volume 43*, pages 1–50. Academic Press, 1996.
5. Nikolaj Bjørner, Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, February 1997.
6. François Bourdoncle. Abstract debugging of higher-order imperative languages. In *SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 46–55, 1993.
7. Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. In Mark Harman and Keith Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 595–607. Elsevier Science B. V., 1998.

8. Gerardo Canfora, Aniello Cimitile, Andrea De Lucia, and G. A. Di Lucca. Software salvaging based on conditions. In *International Conference on Software Maintenance (ICSM'96)*, pages 424–433, Victoria, Canada, September 1994. IEEE Computer Society Press, Los Alamitos, California, USA.
9. Aniello Cimitile, Andrea De Lucia, and Malcolm 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.
10. Aniello Cimitile, Andrea De Lucia, and Malcolm Munro. Qualifying reusable functions using symbolic execution. In *Proceedings of the 2nd working conference on reverse engineering*, pages 178–187, Toronto, Canada, 1995. IEEE Computer Society Press, Los Alamitos, California, USA.
11. Lori A. Clarke, J. Hassell, and D. J. Richardson. A close look at domain testing. *IEEE Transactions on Software Engineering*, 8:380–390, 1982.
12. Alberto Coen-Porisini, Flavio De Paoli, Carlo Ghezzi, and Dino Mandrioli. Software specialization via symbolic execution. *IEEE Transactions on Software Engineering*, 17(9):884–899, September 1991.
13. P. David Coward. Symbolic execution systems - a review. *Software Engineering Journal*, 3(6):229–239, November 1988.
14. Sebastian Danicic, Chris Fox, Mark Harman, and Rob Mark Hierons. ConSIT: A conditioned program slicer. In *IEEE International Conference on Software Maintenance (ICSM'00)*, pages 216–226, San Jose, California, USA, October 2000. IEEE Computer Society Press, Los Alamitos, California, USA.
15. Sebastian Danicic and Mark Harman. Espresso: A slicer generator. In *ACM Symposium on Applied Computing, (SAC'00)*, pages 831–839, Como, Italy, March 2000.
16. Sebastian Danicic, Mark Harman, and Yogasundary Sivagurunathan. A parallel algorithm for static program slicing. *Information Processing Letters*, 56(6):307–313, December 1995.
17. Andrea De Lucia. Private communication, 1996.
18. Andrea De Lucia. Program slicing: Methods and applications. In *1st IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149, Florence, Italy, 2001. IEEE Computer Society Press, Los Alamitos, California, USA.
19. Andrea De Lucia, Anna Rita Fasolino, and Malcolm Munro. Understanding function behaviours through program slicing. In *4th IEEE Workshop on Program Comprehension*, pages 9–18, Berlin, Germany, March 1996. IEEE Computer Society Press, Los Alamitos, California, USA.
20. Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001.
21. Andrei P. Ershov. *On the essence of computation*, pages 391–420. North-Holland, 1978.
22. John Field, G. Ramalingam, and Frank Tip. Parametric program slicing. In *22nd ACM Symposium on Principles of Programming Languages*, pages 379–392, San Francisco, CA, 1995.
23. B. Fischer. Specification-based browsing of software component libraries. In *Thirteenth International Conference on Automated Software Engineering*, pages 74–83. IEEE Computer Society Press, 1998.
24. David W. Flater, Yelina Yesha, and E. K. Park. Extensions to the C programming language for enhanced fault detection. *Software Practice and Experience*, 23(6):617–628, June 1993.
25. Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, Providence, RI, 1967.
26. Chris Fox, Mark Harman, Rob Mark Hierons, and Sebastian Danicic. Backward conditioning: a new program specialisation technique and its application to program comprehension. In *9th IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 89–97, Toronto, Canada, May 2001. IEEE Computer Society Press, Los Alamitos, California, USA.
27. Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler compiler. *Systems, Computers, Controls*, 2(5):721–728, August 1971.
28. Yoshihiko Futamura and Kenroku Nogi. Generalized partial computation. In D. Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, pages 133–151. North-Holland, 1987.
29. Allen Goldberg, T. C. Wang, and David Zimmerman. Applications of feasible path analysis to program testing. In Thomas Ostrand, editor, *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)*, pages 80–94, 1994.
30. Mark Harman, Rob Mark Hierons, Sebastian Danicic, John Howroyd, and Chris Fox. Pre/post conditioned slicing. In *IEEE International Conference on Software Maintenance (ICSM'01)*, pages 138–147, Florence, Italy, November 2001. IEEE Computer Society Press, Los Alamitos, California, USA.
31. Mark Harman and Robert Mark Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.
32. Robert Mark Hierons, Mark Harman, Chris Fox, Lahcen Ouarbya, and Mohammed Daoudi. Conditioned slicing supports partition testing. *Software Testing, Verification and Reliability*, 12:23–28, March 2002.
33. Charles Anthony Richard Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

34. Susan Horwitz, Thomas Reps, and David Wendell Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.
35. Ali Jaoua and Ali Mili. The use of executable assertions for error detection and damage assessment. *Journal of Systems and Software*, (1):15–37, 1990.
36. Mariam Kamkar. *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.
37. Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.
38. Bogdan Korel and Jurgen Rilling. Dynamic program slicing methods. In Mark Harman and Keith Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 647–659. Elsevier, 1998.
39. Wan Kwon Lee, In Sang Chung, Gwang Sik Yoon, and Yong Rae Kwon. Specification-based program slicing and its applications. *Journal of Systems Architecture*, 47:427–443, 2001.
40. Jeremy R. Levitt. *Formal Verification Techniques for Digital Systems*. PhD thesis, Stanford University, December 1998.
41. Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in software development environments. *SIGPLAN Notices*, 19(5):177–184, 1984.
42. Thomas Reps and Todd Turnidge. Program specialization via program slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the Dagstuhl Seminar on Partial Evaluation*, volume 1110, pages 409–429, Schloss Dagstuhl, Wadern, Germany, 12–16 February 1996. Springer-Verlag, New York, NY.
43. D. J. Richardson and L. A. Clarke. Partition analysis: A method combining testing and verification. *IEEE Transactions on Software Engineering*, 14:1477–1490, 1985.
44. David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.
45. Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.
46. Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, Cambridge, England, June 1949. University Mathematical Laboratory.
47. Mark Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.
48. Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.
49. Lee J. White and E. I. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, 6:247–257, 1980.