ConSIT: A Conditioned Program Slicer

Sebastian Danicic,
Goldsmiths College,
University of London,
New Cross,
London SE14 6NW, UK.
Tel: +44 (0)20 7919 7856
Fax: +44 (0)20 7919 7853
sebastian@mcs.gold.ac.uk

Chris Fox,
Kings College,
University of London,
Strand,
London, WC2R 2LS, UK.
Tel: +44 (0)20 7848 2588
Fax: +44 (0)20 7848 2851
foxcj@dcs.kcl.ac.uk

Mark Harman,
Rob Hierons,
Brunel University,
Uxbridge, Middlesex,
UB8 3PH, UK.
Tel: +44 (0)1895 274 000
Fax: +44 (0)1895 251 686
mark.harman@brunel.ac.uk

Abstract

Conditioned slicing is a powerful generalisation of static and dynamic slicing which has applications to many problems in software maintenance and evolution, including re-use, re-engineering and program comprehension.

However, there has been relatively little work on the implementation of conditioned slicing. Algorithms for implementing conditioned slicing necessarily involve reasoning about the values of program predicates in certain sets of states derived from the conditioned slicing criterion, making implementation particularly demanding.

This paper introduces ConSIT, a conditioned slicing system which is based upon conventional static slicing, symbolic execution and theorem proving. ConSIT is the first fully automated implementation of conditioned slicing.

An implementation of ConSIT is available for experimentation at `http://www.mcs.gold.ac.uk/~mas01sd/consit.html`.

1. Introduction

Program slicing is a source level code extraction technique that has been extensively applied to many problems in software maintenance, including, debugging [19, 21, 24, 28], re-engineering [7, 16, 23] and program comprehension [17, 18].

Traditionally, slices have been constructed using either purely static or purely dynamic analysis techniques [1, 20, 22, 29]. The traditional static slicing criterion consists of a pair, (V, n), where V is a set of variables of interest and n is some point of interest in the program. Statements which cannot affect the value of any variable in V when the next statement to be executed is at position n in the program are removed to form the static slice. Figure 1 presents a simple `C` program[1], for which the static slice with respect to the criterion $(28, \{\texttt{sum}\})$ is depicted in Figure 2. The traditional dynamic slicing criterion augments the static criterion with an input sequence, I. Statements which cannot affect the value of any variable in V when the next statement to be executed is at position n *and the input is* I are removed to form the dynamic slice.

Static slices are thus constructed with respect to *no* additional information about the state in which the program is to be executed, while dynamic slices are constructed with respect to complete information about the initial state.

Conditioned slicing is a generalisation of both static and dynamic slicing. The conditioned slicing criterion augments the static criterion with a condition, which captures a set of possible initial states for which the slice and the original program must agree [5]. Definition 1.1 provides a more formal definition of conditioned slice.

Definition 1.1 (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 satisfies π.

For example, Figure 3 shows the conditioned slice constructed from the program in Figure 1 for the criterion $(\{\texttt{sum}\}, end, \forall i.\texttt{a}_i > 0)$, where end is the end of the program and $\forall i.\texttt{a}_i > 0$ indicates that all values read into the variable `a` are positive.

Conditioned slicing is of both theoretical and practical importance. It is theoretically important because it subsumes both static and dynamic slicing [5]. It is practically important because of its application to problems in re-use [7, 8, 9, 10], re-engineering [7, 23] and program comprehension [5, 14, 18].

[1] This example is Due to Canfora, Cimitile and DeLucia [5]. The figure 28 refers to the exit node of the program.

Currently, there is little work on the implementation of conditioned slicing. Existing proof of concept prototype conditioned slicers, such as that described in [5], are interactive. The human is used to answer questions about conditions which arise during the process of analysis [13]. These prototypes serve to illustrate the importance and application of conditioned slicing. However, for conditioned slicing to achieve its potential, fully automated conditioned slicers must be implemented. Such implementations will need to be able to reason about the effect of the conditions mentioned in the slicing criterion. This paper shows how a traditional static slicer can be combined with a theorem prover and symbolic executor to achieve the goal of implementing a fully automated conditioned slicing system. The implementation, ConSIT, produces conditioned slices for an intraprocedural subset of C, the syntax of which is defined in Figure 4.

The rest of this paper is organised as follows: Section 2 describes the implementation of ConSIT, which relies upon symbolic execution and theorem proving subsystems described in Sections 3 and 4 respectively. Section 5 presents two examples of slices produced using ConSIT. The first is the example used by Canfora, Cimitile and DeLucia [5], which is presented as a 'benchmark' against which ConSIT is compared to prior work in this area. The second illustrates the application of conditioned slicing to business rule extraction. Section 6 concludes with directions for future work.

2. The Implementation of ConSIT

The ConSIT system operates on a subset of C, for which a tokeniser and symbolic executor were written in Prolog.

The top level algorithm is quite simple. It is depicted in Figure 5. Phase 1 propagates state information from the condition in the slicing criterion, to all points in the program, using the symbolic execution algorithm described in more detail in Section 3. Phase 2 produces a conditioned program by eliminating statements which are never executed when the initial state satisfies the condition mentioned in the slicing criterion. These are precisely those for which the state information defines an inconsistent set of states. Such statements become unreachable when the program is executed in a state which satisfies the condition of the slicing criterion and are 'sliced away'. The test of consistency of each set of states is computed using the `Isabelle` theorem prover, as described in more detail in Section 4. Phase 3 removes statements from the conditioned program which do not affect the static part of the conditioned slicing criterion. Phase 3 is implemented using the *Espresso* static slicing system [11].

The architecture of the ConSIT system is illustrated in Figure 6. The system is built from various components written in different languages. For the purpose of rapid prototyping, the symbolic executor and conditioner were written in Prolog. These were developed to work on a bespoke imperative programming language called `Haste` that includes loops, input statements and conditionals. The conditions of Haste are defined to be a good match for legal `Isabelle` propositions, and it is easy to parse using a Definite Clause Grammar within Prolog.

```
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);
    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; }
```

Figure 1. Example from Canfora et al [5]

The `Isabelle` theorem prover [26, 27, 25] is used to check the reachability of a statement. This is written in Standard ML. A wrapper script written in `Expect` acts as an `Isabelle` server process. It runs the theorem prover on a pseudo terminal and listens for connections on a socket. When the conditioner requires the services of `Isabelle`, it spawns an `Expect` client process which connects to the `Isabelle` server, via its socket, and requests that the symbolic state be analysed (using `Isabelle`'s auto-tacticals). The server then interacts with `Isabelle` over the pseudo terminal, returning the result of the query back to the client process. The exit code of the client process is then used to indicate the result of the query.

This architecture means that there was no need to develop the symbolic executor within `Isabelle`, and that several processors can make use of `Isabelle` without additional overheads of running several copies of the theorem prover. It also avoids the long delays that would result if the theorem prover process was started every time it was needed. An alternative would be to link the Prolog code with `ML` via `C` interface libraries, but such

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

Figure 2. Static Slice of Figure 1 w.r.t. $(\{\mathtt{sum}\}, end)$

```
main() {
  int a, n, i, possum, negsum, sum;
  scanf("%d", &n);
  i = 1;
  possum = 0;
  negsum = 0;
  while (i <= n) {
    scanf("%d", &a);
    if (a > 0)
      possum = possum + a;
    i=i+1; }
  if (possum >= negsum)
    sum = possum; }
```

Figure 3. Conditioned Slice of Figure 1 w.r.t. $(\{\mathtt{sum}\}, end, \forall i.\mathtt{a}_i > 0)$

a solution is not as portable nor as flexible; with the current architecture the resource intensive theorem prover can be run on a separate high-performance server, and the system is dependent on particular versions of neither Prolog nor ML.

A similar client-server approach is adopted with the slicer. However, in this case the slicer and its server process are written in Java, based upon the *Espresso* slicing system [11]. *Espresso* uses a parallel slicing algorithm [12] which takes advantage of inherent CFG parallelism, where each node of the CFG of the subject program is compiled into a separate Java thread.

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

```
<program>   ::=   main() { <decl-list>
                  <stat-list> }
<decl>      ::=   <type> <var-list> ;
<decl-list> ::=   <decl> <decl-list> | <empty>
<var-list>  ::=   <var> | <var> , <var-list>
<stat>      ::=   { <stat-list> } |
                  <var> = <expr>; |
                  scanf(<string>,<amp-list>); |
                  if (<expr>) <stat>  |
                  if (<expr>) <stat>
                  else <stat> |
                  while (<expr>) <stat> |
                  ASSERT(<expr>) ;   |
                  <empty> ;
<stat-list>  ::=  <stat> <stat-list> | <empty>
<empty>      ::=
<string>     ::=  "<char-list>"
<amp-list>   ::=  &<var> | &<var>, <amp-list>
<expr>       ::=  <unaryop> <expr> |
                  <expr> <binop> <expr> |
                  <var> | <integer-constant> |
                  ( <expr> ) | <char-constant>
<binop>      ::=  < | > | <= |>= | == | != |
                  || | && | + | * | - | /
<unaryop>    ::=  - | !
<char-const> ::=  '<char>'
```

Figure 4. The subset of C accepted by ConSIT

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

Figure 5. The Three Phases in Slice Construction

The slightly contrived acronym ConSIT stands for **Con**ditioned **S**licer using the **I**sabelle **T**heorem prover. The acronym is chosen because ConSIT combines several different systems and languages in concert.

2.1. Representing the Slicing Criterion

ConSIT uses a novel technique for representing the slicing criterion and presenting it to the system, which the authors have found to be very effective and worthy of mention. The slicing criterion is effectively *encoded* into the source code of the program to be sliced.

The Conditioned Part of the Criterion
The condition is inserted into the program using an ASSERT statement, which takes a single boolean argument and asserts that its value is true, allowing considerable flexibility in constructing a slicing criterion. This is because the ASSERT statement can be inserted at *any* point in the program.

To perform a conditioned slice the ASSERT statement will

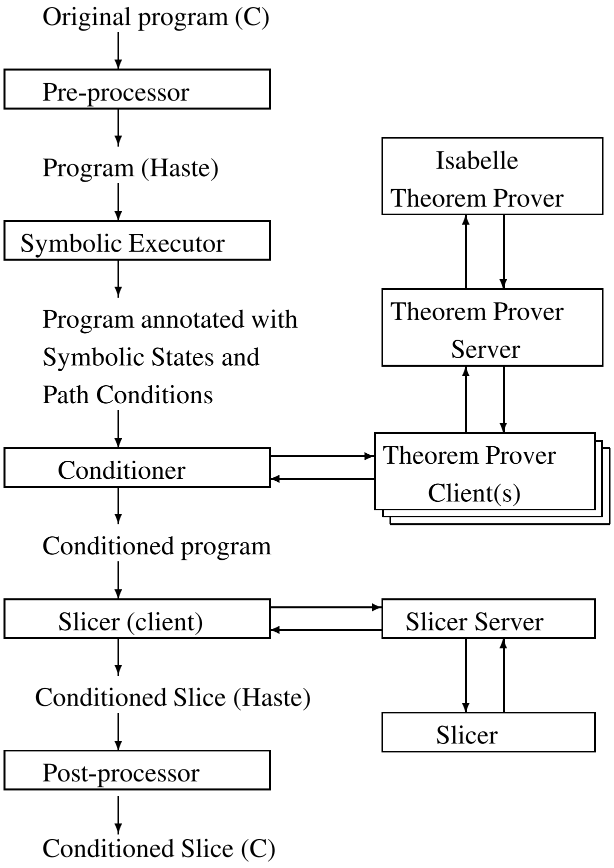

Figure 6. The Architecture of ConSIT

typically be added to the beginning of the program. However, there is no reason why `ASSERT` statements may not be added to arbitrary parts of the program.

The `ASSERT` statement is also used to handle conditions on input variables. When defining a conditioned slicing criterion, input variables present a problem because the value of the variable may be read in a loop, causing it to have many values during the execution of the program. Canfora et al [5] address this issue by subscripting the variable name, allowing universal quantification and ranges to be used. For example, the condition in the slicing criterion for the example program in Figure 1 is $\forall i.1 \leq i \leq \mathtt{n}.\mathtt{a}_i > 0$, specifying that all values read into the variable `a` are positive.

Unfortunately, this approach becomes inconvenient when it is not easy to specify the range of subscripts of interest. For example, in the code fragment below, it becomes hard to specify that all values read into the variable `a` by the *second* `while` loop are positive.

```
while (i>0) {
  scanf("%d",&a) ;
  ...
  i = i-1;
  }
while (j>0) {
  scanf("%d",&a) ;
  ...
  j = j - 1;
  }
```

That is, suppose that the initial value of the variables `i` and `j` are dependent upon the input in some arbitrarily complex way. It will not be obvious how many times the first loop executed and therefore difficult to delimit the subscripts which should be applied to the variable `a` to capture the set of 'occurrences' of interest.

Using the `ASSERT` statement, this becomes very easy. All that is required, is to add an `ASSERT` in the second loop, giving:-

```
while (i>0) {
  scanf("%d",&a) ;
  ...
  i = i-1;
  }
while (j>0) {
  scanf("%d",&a) ;
  ASSERT(a>0) ;
  ...
  j = j - 1;
  }
```

¿ From a theoretical point of view, the use of the `ASSERT` statement does not make for a more general form of conditioned slicing; all `ASSERT` statements could, theoretically, be moved to the beginning of the program using predicate transformation similar to that proposed by Dijkstra for program verification [15]. However, the `ASSERT` statement allows for the specification of local constraints in input variables and is thus very helpful from a practical point of view in specifying conditioned slicing criteria.

The Static Part of the Criterion

The static part of the slicing criterion is also inserted directly into the program as program code. Recall that the static part of the criterion consists of the set of variables of interest, V and the point at which those variables' values are of interest, n. The static part of the criterion is captured by a program variable `slice`. The user inserts an assignment to this variable at an arbitrary set of points in the program. If the assignment

$$\mathtt{slice} = \mathtt{f}(x, y, z)$$

is inserted at point n in the program, then the slice must preserve the values of x, y and z at point n in the program. This notation allows for multi-slicing [12, 29], in which there may be arbitrarily many slice points in the program, each of which is concerned with a different set of variables of interest.

3. The Symbolic Execution Phase

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

In practice, symbolic states are represented as sets of *pairs*. The first element of each pair is a set of bindings between variables and their symbolic values that arise from assignments and input statements. Canfora et al [5] call this the *symbolic state*. The second element of each pair is a set of (linear) inequalities. Canfora et al [5] call this the *path condition*. The intended interpretation is that each pair corresponds to a conditional symbolic state; the variables will have the symbolic values given in the first element of the pair when the (in)equalities in the second element are true. This distinction between assignment of value and other conditions is helpful when dereferencing a variable.

The symbolic executor is derived directly from the semantics of the language. The semantics $\sigma(s)$ of a sequence of statements s is a set of states $\{s_1, s_2, \ldots, s_n\}$. Each symbolic state s_i ($1 \leq i \leq n$) can be considered as the pair[2]:

$$\langle a_i, c_i \rangle$$

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

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

3.1. Arithmetic and Boolean Expressions

Functions are defined to evaluate expressions given an assignment function. The definition of these functions simply propagates the function through the syntax of the expressions in a syntax directed manner. Variables are replaced by their current symbolic values in the state g. Some example clauses from the definition of these two functions are given below:

$$\begin{array}{lcl}
E_g(\mathit{var}) & = & g(\mathit{var}) \\
E_g(\mathit{const}) & = & \mathit{const} \\
E_g(-\mathit{expr}) & = & -E_g(\mathit{expr}) \\
E_g(\texttt{(}\mathit{expr}_1\texttt{+}\mathit{expr}_2\texttt{)}) & = & (E_g(\mathit{expr}_1) + E_g(\mathit{expr}_2)) \\
E_g(\texttt{(}\mathit{expr}_1\texttt{-}\mathit{expr}_2\texttt{)}) & = & (E_g(\mathit{expr}_1) - E_g(\mathit{expr}_2)) \\
E_g(\texttt{(}\mathit{expr}_1\texttt{*}\mathit{expr}_2\texttt{)}) & = & (E_g(\mathit{expr}_1) \times E_g(\mathit{expr}_2))
\end{array}$$

$$\begin{array}{lcl}
B_g(\texttt{(}\mathit{expr}_1\texttt{==}\mathit{expr}_2\texttt{)}) & = & (E_g(\mathit{expr}_1) = E_g(\mathit{expr}_2)) \\
B_g(\texttt{(}\mathit{expr}_1\texttt{<}\mathit{expr}_2\texttt{)}) & = & (E_g(\mathit{expr}_1) < E_g(\mathit{expr}_2)) \\
B_g(\texttt{!}\ \mathit{cond}) & = & \neg B_g(\mathit{cond}) \\
B_g(\texttt{(}\mathit{cond}_1\texttt{||}\mathit{cond}_2\texttt{)}) & = & (B_g(\mathit{cond}_1) \vee B_g(\mathit{cond}_2)) \\
B_g(\texttt{(}\mathit{cond}_1\texttt{\&\&}\mathit{cond}_2\texttt{)}) & = & (B_g(\mathit{cond}_1) \wedge B_g(\mathit{cond}_2))
\end{array}$$

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

3.2. Statements

It is now possible to give a recursive definition of $\sigma(s)$. In general, a statement is executed in the context of a symbolic assignment function; the meaning of a statement depends upon the symbolic values that have been assigned to any defined program variables. In the following, this is modelled with a contextual effect using a λ-abstracted assignment function g. All but one of the following definitions interprets the meaning of a program as a function that expects an assignment function as its argument. The exception is at the top-most level of the program, where no values have been assigned to any variables. The meaning of a program s is thus the meaning of s applied to the empty assignment function $\emptyset$:

$$\sigma(s)\emptyset$$

Block Statements

The meaning of a block is the meaning of the sequence of statements it contains.

$$\sigma(\texttt{\{}\ s\ \texttt{\}}) = \sigma(s)$$

Empty Statements

An empty statement gives rise to an empty assignment function, and an empty condition set.

$$\sigma(\texttt{;}) = \lambda g.\{\langle \emptyset, \emptyset \rangle\}$$

Assignment Statements

An assignment *var=expr* produces an assignment function consisting of variable *var* and the expression *expr* evaluated in the context of the current assignment function g. No conditions are created.

$$\sigma(\mathit{var}\texttt{=}\mathit{expr}) = \lambda g.\{\langle \{\langle \mathit{var}, E_g(\mathit{expr}) \rangle\}, \emptyset \rangle\}$$

Input Statements

When a `scanf("%d",&x);` statement is evaluated, nothing can be assumed about the value input to the variable `x`. This is modelled by setting `x` to some unique symbolic value that, in the semantics, is intended to be interpreted as an existentially quantified variable. That is, the value of the variable is set to some unique skolem constant. It will be unique in the sense that it does not appear in the interpretation of the program leading up to that `scanf` statement. For this purpose, the term $sk_g^?$ will be used to refer to an "unused" skolem constant with respect to an assignment function g. That is $sk_g^?$ will be an element of $\{sk_n : n \in \mathbb{Z}^+ \wedge \neg\exists v. \langle v, sk_n \rangle \in g\}$.

$$\sigma(\texttt{scanf("\%d",\&}\mathit{var}\texttt{)}) = \lambda g.\{\langle \{\langle \mathit{var}, sk_g^? \rangle\}, \emptyset \rangle\}$$

Conditional Statements

In the case of a conditional, the symbolic execution results in two paths, one where the condition is `true`, and the `then` path is taken, and the second where the condition is `false`, and the `else` path is taken.

$$\begin{aligned}&\sigma(\texttt{if}\,(\ \mathit{cond}\ )\ s_1\ \texttt{else}\ s_2)\\ &\quad= \lambda g.(\{\langle a, c \cup \{B_g(\mathit{cond})\}\rangle : \langle a,c\rangle \in \sigma(s_1)g\} \cup\\ &\qquad\quad \{\langle a, c \cup \{\neg B_g(\mathit{cond})\}\rangle : \langle a,c\rangle \in \sigma(s_2)g\})\end{aligned}$$

Statement Sequences

In the symbolic semantics of sequences of statements, $s_1\,;\,s_2$, each statement gives rise to a set of pairs of symbolic assignments and path conditions. Considering all the permutations, sequencing two statements results in a set of conditional statements where the total number of pairs of assignments and path conditions is the product of the number of conditional states in the symbolic execution of the constituent statements. As with individual statements, each of the conditional states will be a pair consisting of an assignment function and a path condition. For each combination of elements $\langle a_1, c_1\rangle$ of $\sigma(s_1)$ in the context of an assignment function g, and elements $\langle a_2, c_2\rangle$ of $\sigma(s_2)$ in the context of an assignment function g updated by the assignment function a_1 (formally, $g \oplus a_1$), there will be an element of $\sigma(s_1\,;\,s_2)$ consisting of a_1 updated by a_2, paired with the union of c_1 and c_2: $\langle a_1 \oplus a_2, c_1 \cup c_2\rangle$. The assignment update operator $\oplus$ is defined by

$$f \oplus h = \{\langle v, e\rangle : \langle v, e\rangle \in h \vee (\langle v, e\rangle \in f \wedge \neg\exists e'.\langle v, e'\rangle \in h)\}$$

Using this, we can formalise the semantics of statement sequencing as

$$\begin{aligned}\sigma(s_1\,;\,s_2) = \lambda g.\{\langle a_1 \oplus a_2, c_1 \cup c_2\rangle :\ & \\ \langle a_1, c_1\rangle \in \sigma(s_1)g \wedge\ & \\ \langle a_2, c_2\rangle \in \sigma(s_2)(g \oplus a_1)\}&\end{aligned}$$

`WHILE` Statements

With `while` loops, there are various ways of computing a symbolic execution. Perhaps one extreme is just to reset all variables that are assigned to so that it is not possible to infer anything about their values on termination of the loop, other than the fact that the termination condition is met. The other extreme would be to try and build some fixed point construction that models the `while` loop. The first suffers from being excessively weak; no loop invariant could be determined, for example. For the application to conditioned slicing, the second extreme suffers from being excessively complicated; although all relevant information might be encoded in the fixed point construction it is not readily accessible to the automatic theorem proving phase required to infer (relatively simple) theorems concerning boolean predicates.

The ConSIT system adopts a compromise which lies somewhere in between the two extremes. Observe that, either a loop is executed at least once, or not at all. The latter case is trivial, and is accounted for by adding the negation of the loop condition to each of the current conditional states. In the former case, if the loop terminates, it may have been executed more than once prior to termination, and so the system cannot readily determine the values of any variables. However, it can give the values of the variables derived on the final execution of the loop as a function of the previous values of these variables, although it may have to represent these previous values as being unknown, since they might have been changed by previous executions of the loop body. The loop condition is `true` at the beginning of each iteration, and `false` immediately after the loop.

To implement this, the system uses the set of variables which are defined in the body of the loop. More formally, the set of defined variables, $\nu(s)$ of a statement s is defined as follows

$$\nu(s) = \{v : \exists e\alpha\kappa.(\langle v, e\rangle \in \alpha \wedge \langle \alpha, \kappa\rangle \in \sigma(s)\emptyset)\}$$

The defined variables are those whose values might have been changed on prior executions of the loop. In the symbolic execution, we can take their unknown penultimate values to be unique skolem constants. A unique set of skolemisations, U^g_{sk}, of a list of variables, $[v_1, v_2, \ldots, v_n]$, in the context of an assignment g, is given by $\{\langle v_1, sk^?_{g_1}\rangle, \langle v_2, sk^?_{g_2}\rangle, \ldots, \langle v_n, sk^?_{g_n}\rangle\}$ where

1. $g_1 = g$
2. $g_i = g_{i-1} \oplus \{\langle v_{i-1}, sk^?_{g_{i-1}}\rangle\}$

where $\oplus$ is the previously defined assignment update operator.

Using U^g_{sk}, a unique skolemisation of statements s is given by:

$$U^g_{sk}(O(\nu(s)))$$

where O imposes some total ordering on the set of variables $\nu(s)$, to give a list of variables.

The symbolic states obtained by a `WHILE` loop are then the union of

1. The symbolic states prior to the `while` loop, with the additional constraint that the loop condition is `false` in those states.
2. The symbolic states that result from the execution of the loop body, in the context where all defined variables are uniquely skolemised, with the additional constraint that the loop condition was `true` in the initial states, and `false` in the final states.

$$\begin{aligned}&\sigma(\texttt{while}\,(\ \mathit{cond}\ )\ s)\\ &\quad= \lambda g.(\{\langle\emptyset, \{\neg B_g(\mathit{cond})\}\rangle\} \cup\\ &\qquad\quad \{\langle a, c \cup \{B_g(\mathit{cond}), \neg B_a(\mathit{cond})\}\rangle :\\ &\qquad\qquad \langle a, c\rangle \in \sigma(s)(g \oplus U^g_{sk}(O(\nu(s))))\})\end{aligned}$$

4. The Theorem Proving Phase

The symbolic execution phase takes a sequence of program statements and annotates it with symbolic state descriptions. The implementation seeks to simplify these symbolic states by eliminating those conditional states that are inconsistent from each symbolic state description to determine the paths through the structure of the program code that can never be taken. In 'normal' programs, these paths are unlikely to arise as they would constitute 'bad programming'. However, in conditioned slicing, the program is sliced with respect to an initial condition and the whole point of conditioning is to identify parts of the code which become unreachable as a result. (These are sliced away as irrelevant to the slicing criterion in the same way that statements which have no effect upon the chosen variables are sliced away.)

More precisely, if a conditional state is universally valid, then it will be the only state that can be reached. If all the conditional states of a symbolic state are inconsistent, then that point in the program will be unreachable. The theorem prover is thus used to determine whether the outcome of a predicate *must* be `true` or whether it must be `false` or whether it is not possible to tell. This process is inherently conservative, because there will be predicates which must be `true` and those which must be `false` but for which this information cannot be deduced by the theorem prover. However, this conservatism is safe: if a statement is removed because of the outcome of the theorem proving stage, then that statement is guaranteed to be unnecessary in all states which satisfy the initial condition mentioned in the conditioned slicing criterion.

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

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

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

and

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

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

4.1. Checking for Consistency using `Isabelle`

Automatic theorem proving can be of several flavours. The simplest form might repeatedly apply a simple inference rule to the negation of the statement to be proved in the hope that eventually, a contradiction will arise. This can be very inefficient. The kinds of statements that need to be proved involve inequalities. Such transitive relations can lead to a large search space, making computations hopelessly slow.

`Isabelle` adopts a different approach, where more complicated inference rules can be applied in the form of *tactics*. A tactic is applied. If it fails to have the desired effect, then it is undone and a different tactic is attempted in much the same way as program transformation systems apply transformation tactics [3]. In a sense, `Isabelle` can be regarded as an aid to rigorous theorem proving. A collection of tactics is combined into a strategy, or *tactical*. In this application, tactical are used which attempt to perform the consistency check fully automatically.

In the initial system that was developed, a range of different tacticals had to be applied to solve the inequality puzzles. In some cases, it was necessary to temporarily reorder the tactics within a tactical. Given the obvious potential pitfalls of automating the theorem proving with `Isabelle`, the system was designed to be robust; if a theorem cannot be proven, or should an internal `Isabelle` error occur, the conditioning of the program continues, although it might miss a potential simplification. This gives rise to a safe construction of a conditioned slice, but one which is conservative. Nonetheless, as the examples in Section 5 illustrate, ConSIT is capable of reasonable program simplification.

In cases where a boolean condition is neither `true` nor `false`, but contingent upon the values of its variables, `Isabelle` can sometimes derive a simplification of the condition. Future versions of ConSIT may exploit this, substituting the simplified condition when conditioning the program. The intention of this work is to develop amorphous slicing capability [4, 17].

5. Examples

5.1. The Example from Canfora, Cimitile and DeLucia

Consider again, the example program in Figure 1. Figure 3 shows the conditioned slice produced by the ConSIT system. This is identical to the conditioned slice produced by Canfora, Cimitile and DeLucia using a mixture of automated slicing and manual, human analysis to evaluate conditions.

5.2. Checking and Extracting Business Rules

Conditioned slicing can be used as a technique for business rule extraction. For example, consider the simple tax calculation program in Figure 7.

This program represents a computation of tax codes and amounts of tax payable, including allowances for a United Kingdom citizen in the tax year April 1998 to April 1999. Each person has a personal allowance which is an amount of un-taxed income. The personal allowance depends upon the status of the person, reflected by the boolean variables `blind`, `married` and `widowed` and the integer variable `age`. There are three tax bands, for which tax is charged at the rates of 10%, 23% and 40%. The width of the 10% tax band is subject to the status of the person, while the 23% and 40% are fixed for all individuals.

```
main() {
int age, blind, widow, married, income

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;

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';
}
```

Figure 7. UK Income Taxation Calculation Program

This set of taxation rules constitutes a governmental 'business system', and the program in Figure 7 represents an attempt to capture these rules in program code.

Conditioned slicing allows us to extract from this program, fragments which correspond to certain taxation scenarios. In so doing, the conditioned slicer is identifying the portions of the code which implement individual business rules. For example, using conditioned slicing, it becomes possible to ask

> "what is the personal allowance calculation for a blind widow aged over 68?"

Slicing extracts from the program only the code concerned with computation on `personal` and from this only that code which is relevant to computations which satisfy the scenario in which the person concerned is blind, over 68 years of age and widowed. Observe that as a *range* of possible inputs is specified by this condition, it is not possible to simply run the program in order to arrive at the answer.

For the blind widow over the age of 68, the conditioned slicing computed by ConSIT is given in Figure 8.

```
scanf("%d",&age);
scanf("%d",&blind);

if (age>=75) personal = 5980;
else if (age>=65) personal = 5720;

if (age>=65 && income >16800)
{ t = personal - ((income-16800)/2);
  if (t>4335) personal = t ;
  else personal = 4335; }

if (blind) personal = personal + 1380 ;
```

Figure 8. Specialised Personal Allowance

```
scanf("%d",&age);
scanf("%d",&blind);
scanf("%d",&married);

if (!blind && !married && age < 65)
code = 'L' ;
else if (!blind && age < 65 && married)
code = 'H';
else if (age >= 65 && age < 75
         && !married && !blind)
else if (age >= 65 && age < 75
         && married && !blind)
else code = 'T';
```

Figure 9. Tax Codes for the Under 60s

For similar rule-based systems in which source-code effectively captures business rules, this example suggests that conditioned slicing might be effective in recovering business rules in a similar way to the technique proposed by Canfora et al [6].

Another example of a question that can be addressed using ConSIT, is

> "What are the possible tax codes that apply to people under the age of 60?"

This computation is captured by the conditioned slice depicted in Figure 9.

The example clearly illustrates the program comprehension advantages of removing 'unnecessary predicates' from a conditioned slice. In deeply nested conditional structures it is helpful to remove these unnecessary predicate tests, so that those tests which play a part in the computation of interest become more obvious. Doing this produces the further refined conditioned slicing in Figure 10. The slice in Figure 10 satisfies the definition of a conditioned slice and is simpler than one which retains the predicates, and so such a slice would appear to be better from both a theoretical and a practical point of view. The current implementation does not remove these predicates for compatibility with the work of Canfora et al [5].

```
scanf("%d",&age);
scanf("%d",&blind);
scanf("%d",&married);

if (!blind && !married && age < 65)
code = 'L' ;
else if (!blind && age < 65 && married)
code = 'H';
else code = 'T';
```

Figure 10. Under 60s Without Unnecessary Predicates

6. Conclusion and Future Work

This paper has described ConSIT, a conditioned slicing system which uses symbolic execution, traditional static slicing and the `Isabelle` theorem prover to implement conditioned slicing for a simple intraprocedural subset of `C`.

The symbolic executor propagates state information to each statement in the program. This information is passed to the theorem prover, which determines (in a conservative and safe manner) which statements have become unreachable under the conditions imposed by the slicing criterion. This forms a conditioned program which is further sliced using traditional static slicing.

ConSIT is the first fully automated implementation of conditioned slicing. However, the language handled by ConSIT is somewhat limited. Pointers and procedures are the most notable omissions. Work is in progress to extend ConSIT to handle fully interprocedural conditioned slicing.

Another problem for future work is the adoption of the technologies used to implement ConSIT so that amorphous conditioned slices [4, 17, 19] can be computed. Amorphous slices share the semantic restriction that they preserve the effect of the subject program on the slicing criterion, but are syntactically unrestricted. In cases where the `Isabelle` theorem prover was unable to decide the truth or otherwise of propositions put to it, it was often able to simplify the proposition as a side effect of its computation. This simplification has a direct counterpart in simplification of arithmetic and boolean expressions which would be useful in an implementation of amorphous conditioned slicing.

Isabelle is a very powerful and general purpose theorem prover. This means that it can take considerable time to produce amorphous slices using Isabelle as the engine for deciding propositions. For instances, the first example presented in this paper, takes about 20 minutes to slice on a Pentium II running at 233MHz. While this may be acceptable for a proof of concept implementation, it is clearly inadequate as a basis for extensive development. However, the kind of propositions which arise in conditioned slicing tend to be rather simple and it could be that great speed improvement can be achieved with a more narrowly focussed technology. To this end, the authors plan to investigate the use of other tools for analysis of conditions, such as the Stanford Validity Checker (SVC) [2]. The hope is that the use of tools like the SVC will dramatically improve performance.

Acknowledgements

The authors would like to thank Andrea DeLucia and Gerardo Canfora for several helpful discussions concerning conditioned slicing. Lawrence Paulson and Tobias Nipkow kindly made available a development version of `Isabelle` which contained extra features of use in developing ConSIT. This work is supported in part by EPSRC grants GR/M58719 and GR/M78083.

References

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

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

[3] K. Bennett, T. Bull, E. Younger, and Z. Luo. Bylands: reverse engineering safety-critical systems. In *IEEE International Conference on Software Maintenance*, pages 358–366. IEEE Computer Society Press, Los Alamitos, California, USA, 1995.

[4] D. W. Binkley. Computing amorphous program slices using dependence graphs and a data-flow model. In *ACM Symposium on Applied Computing*, pages 519–525, The Menger, San Antonio, Texas, U.S.A., 1999. ACM Press, New York, NY, USA.

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

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

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

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

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

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

[11] S. Danicic and M. Harman. Espresso: A slicer generator. In *ACM Symposium on Applied Computing, (SAC'00)*, page To appear, Como, Italy, Mar. 2000.

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

[13] A. De Lucia. Private communication, 1999.

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

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

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

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

[18] M. Harman, C. Fox, R. M. Hierons, D. Binkley, and S. Danicic. Program simplification as a means of approximating undecidable propositions. In 7^{th} *IEEE International Workshop on Program Comprehesion (IWPC'99)*, pages 208–217, Pittsburgh, Pennsylvania, USA, May 1999. IEEE Computer Society Press, Los Alamitos, California, USA.

[19] M. Harman, Y. Sivagurunathan, and S. Danicic. Analysis of dynamic memory access using amorphous slicing. In *IEEE International Conference on Software Maintenance (ICSM'98)*, pages 336–345, Bethesda, Maryland, USA, Nov. 1998. IEEE Computer Society Press, Los Alamitos, California, USA.

[20] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[21] M. 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.

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

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

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

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

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

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

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

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