Conditioned Semantic Slicing via Abstraction and Refinement in FermaT

M. P. Ward, H. Zedan and T. Hardcastle
Software Technology Research Lab
De Montfort University
The Gateway,
Leicester LE1 9BH, UK
`Martin.Ward@durham.ac.uk` and $\{$`zedan,timh`$\}$`@dmu.ac.uk`

Abstract

In this paper we describe an improved formalisation of slicing in WSL (Wide Spectrum Language) transformation theory and apply the result to give syntactic and semantic slices for some challenging slicing problems. Although there is no algorithm for constructing a minimal syntactic *slice, we show that it is possible, in the WSL language, to derive a minimal* semantic *slice for any program and any slicing criteria. We describe the* Representation Theorem *and show how it is (partially) implemented in the FermaT transformation system. The theorem has applications to semantic (or conditioned) slicing, and we use a combination of abstraction (via the representation theorem), simplification and refinement plus other program transformations to develop a powerful conditioned slicing algorithm.*

1. Introduction

Program slicing is a decomposition technique that extracts from a program those statements relevant to a particular computation. Informally, a slice provides the answer to the question "What program statements potentially affect the value of variable v at statement s?" An observer cannot distinguish between the execution of a program and execution of the slice, when attention is focused on the value of v in statement s.

Slicing was first described by Mark Weiser [14] as a debugging technique [15], and has since proved to have applications in testing, parallelization, integration, software safety, program understanding and software maintenance. Survey articles by Binkley and Gallagher [1] and Tip [8] include extensive bibliographies.

In [10] a formalisation of slicing in terms of program transformations was proposed. In this paper we present an improved formalisation and apply it to some particularly challenging slicing problems. The WSL (Wide Spectrum Language [9]) formulation of slicing immediately lends itself to several extensions: simply by relaxing some of the constraints in the definition and removing restrictions on the allowed transformations.

Weiser defined a program slice **S** as a *reduced, executable program* obtained from a program **P** by removing statements, such that **S** replicates part of the behavior of **P**. A *program transformation* is any operation on a program which generates a semantically equivalent program. A slice is not generally a transformation of the original program since a transformation has to preserve the *whole* behaviour of the program, while in the slice some statements which affect the values of some output variables (those not in the slice) may have been deleted.

Suppose we are slicing on the *end* of the program: then the subset of the behaviour we want to preserve is simply the final values of one or more variables (the variables in the slicing criterion). If we modify both the original program and the slice to delete the unwanted variables from the state space, then the two modified programs *will* be semantically equivalent. Consider this simple example:

$x := y + 1;$
$y := y + 4;$
$x := x + z$

where we are interested in the final value of x. The assignment to y can be sliced away:

$x := y + 1;$
$x := x + z$

These two programs are not equivalent, because they give different values to y, but if we modify both programs by appending a **remove**(y) statement (to remove y from the final state space) then the resulting programs *are* equivalent.

To be precise, let $\mathbf{S}_1$ be the program:

$x := y + 1;$
$y := y + 4;$

```
$x := x + z$;
remove($y$)
```

and let $\mathbf{S}_2$ be the program:

```
$x := y + 1$;
$x := x + z$;
remove($y$)
```

The initial state space for $\mathbf{S}_1$ and $\mathbf{S}_2$ is $\{x, y, z\}$ while the final state space is $\{x\}$ (the **remove** statement ensures that y cannot appear in the final state space). Both programs set x to the value $y+1+z$, so the two programs are equivalent.

So much for slicing at the end of a program. Suppose we want to slice on the value of i at the top of the **while** loop body in this program:

```
$i := 0$;  $s := 0$;
while $i < n$ do
  $s := s + i$;
  $i := i + 1$ od;
$i := 0$
```

Slicing on i at the end of the program would give $i := 0$ as a valid slice: which is not what we wanted! So we need some way to get the sequence of values taken on by i at the top of the loop to be "carried" to the end of the program. A simple way to do this is to add a new variable, slice, which records this sequence of values:

```
$i := 0$;  $s := 0$;
while $i < n$ do
  slice := slice ++ $\langle i \rangle$;
  $s := s + i$;
  $i := i + 1$ od;
$i := 0$;
```

where the statement slice := slice ++ $\langle i \rangle$ appends the value of i to the end of the sequence stored in slice. By the end of the program, slice contains a list of all the values taken on by i at each iteration of the loop.

Slicing on slice at the end of the program is therefore equivalent to slicing on i at the top of the loop. If we add the statement **remove**(i, s, n) to remove all the variables other than slice, then the result can be transformed into the equivalent program:

```
$i := 0$;
while $i < n$ do
  slice := slice ++ $\langle i \rangle$;
  $i := i + 1$ od;
remove($i, s, n$)
```

which yields the sliced program:

```
$i := 0$;
while $i < n$ do
  $i := i + 1$ od
```

This approach easily generalises to slicing on a set of locations with the same or a different set of variables of interest at each location.

2. Slicing as a Program Transformation

The WSL language has been described elsewhere [9, 13], so will not be described in detail here. Key points to note are that WSL is based on *infinitary* logic: which means that formulae in WSL programs can be infinitely long. The weakest precondition of a WSL program **S**, for a given postcondition, **R**, denoted WP$(\mathbf{S}, \mathbf{R})$, is the weakest condition on the initial state such that if **S** is started in a state which satisfies WP$(\mathbf{S}, \mathbf{R})$ then it is guaranteed to terminate and every possible final state will satisfy **R**. If the postcondition **R** is defined as an infinitary logic formula, then WP$(\mathbf{S}, \mathbf{R})$ can be defined as an infinitary logic formula. Hence, the WP for one program can be used as an assertion or condition in another program.

WSL includes loops of the form **do** ... **od** which can only be terminated via an **exit**(n) statement. The statement **exit**(n) (in which n is an integer, not a variable or expression) causes the immediate termination of n enclosing nested **do** ... **od** loops. A statement in which each **exit**(n) is enclosed in at least n nested loops is called a *proper sequence*: such a statement cannot terminate an enclosing loop.

2.1. Reduction

To give a formal definition of slicing in WSL we need to define a *reduction* of a program. Informally, a reduction of a program has certain statements replaced by **skip** or **exit** statements. We define the relation $\mathbf{S}_1 \sqsubseteq \mathbf{S}_2$, read "$\mathbf{S}_1$ is a reduction of $\mathbf{S}_2$", on WSL programs as follows:

$$\mathbf{S} \sqsubseteq \mathbf{S} \quad \text{for any program } \mathbf{S}$$

$$\textbf{skip} \sqsubseteq \mathbf{S} \quad \text{for any proper sequence } \mathbf{S}$$

If **S** is not a proper sequence and $n > 0$ is the largest integer such that there is an **exit**$(n+k)$ within $k \geqslant 0$ nested **do** ... **od** loops in **S**, then:

$$\textbf{exit}(n) \sqsubseteq \mathbf{S}$$

(In other words, if execution of **S** could result in the termination of at most n enclosing loops, then **S** can be reduced to **exit**(n)).

If $\mathbf{S}'_1 \sqsubseteq \mathbf{S}_1$ and $\mathbf{S}'_2 \sqsubseteq \mathbf{S}_2$ then:

$$\textbf{if B then } \mathbf{S}'_1 \textbf{ else } \mathbf{S}'_2 \textbf{ fi} \sqsubseteq \textbf{if B then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi}$$

If $\mathbf{S}' \sqsubseteq \mathbf{S}$ then:

$$\textbf{while B do S}' \textbf{ od} \sqsubseteq \textbf{while B do S od}$$

$$\textbf{var } \langle v := e \rangle : \mathbf{S}' \textbf{ end} \sqsubseteq \textbf{var } \langle v := e \rangle : \mathbf{S} \textbf{ end}$$

$$\textbf{var } \langle v := \perp \rangle : \mathbf{S}' \textbf{ end} \sqsubseteq \textbf{var } \langle v := e \rangle : \mathbf{S} \textbf{ end}$$

where $\perp$ represents the undefined value. This last case will be used when the variable v is used in **S**, but the initial value e is not used.

If $\mathbf{S}'_i \sqsubseteq \mathbf{S}_i$ for $1 \leqslant i \leqslant n$ then:

$$\mathbf{S}'_1;\ \mathbf{S}'_2;\ \ldots;\ \mathbf{S}'_n \sqsubseteq \mathbf{S}_1;\ \mathbf{S}_2;\ \ldots;\ \mathbf{S}_n$$

The reduction relation does not allow deletion of a **skip** statement from a sequence of statements. This is so that the position of each subcomponent of the reduced program is the same as the corresponding subcomponent in the original program. This relationship makes it easer to prove the correctness of slicing algorithms: deleting the extraneous **skip** statements is a trivial additional step. In what follows we will omit the extra **skip** statements when the relationship between the original and sliced programs is clear.

Three important properties of the reduction relation are:

Lemma 2.1 Transitivity: If $\mathbf{S}_1 \sqsubseteq \mathbf{S}_2$ and $\mathbf{S}_2 \sqsubseteq \mathbf{S}_3$ then $\mathbf{S}_1 \sqsubseteq \mathbf{S}_3$.

Lemma 2.2 Antisymmetry: If $\mathbf{S}_1 \sqsubseteq \mathbf{S}_2$ and $\mathbf{S}_2 \sqsubseteq \mathbf{S}_1$ then $\mathbf{S}_1 = \mathbf{S}_2$.

Lemma 2.3 The *Replacement Property*: If any component of a program is replaced by a reduction, then the result is a reduction of the whole program.

2.2. Syntactic Slicing

In [10] a syntactic slice was defined as any reduction of the program which is also a *refinement* of the program. This definition allows a program slicer to delete loops which do not affect the variables in the slicing criteria without having to prove termination of the loop (most slicing researchers allow deletion of nonterminating code as a valid slice). But such a definition of slicing is counter-intuitive in the sense that slicing is intuitively an *abstraction* operation (an operation which throws away information) while refinement is the opposite of abstraction. A more important consideration is that we would like to be able to analyse the sliced program and derive facts about the original program (with the proviso that the original program might not terminate in cases where the slice does). If the sliced program assigns a particular value to a variable in the slice, then we would like to deduce that the original program assigns the *same* value to the variable. But with the refinement definition of a slice, the fact that the slice sets x to 1, say, tells us only that 1 is one of the *possible* values given to x by the original program.

Consider the following nondeterministic program which we want to slice on the final value of x:

```
$x := 1$;
while $n > 1$ do
  if even?$(n)$ then $n := n/2$
               else $n := 3 * n + 1$ fi od;
if true $\rightarrow x := 1$
$\Box$ true $\rightarrow x := 2$ fi
```

The **while** loop clearly does not affect x, so we would like to delete it from the slice. But if we are insisting that the slice be *equivalent* to the original program (on x), then we have to prove that the loop terminates for all n before we can delete it. The loop generates the Collatz sequence and it is an open question as to whether the sequence always reaches 1. (The problem was first posed by L. Collatz in 1937 [6,7]).

Allowing any refinement as a valid slice (as in [10]) would allow us to delete the **while** loop, but would also allow us to delete the **if** statement, giving $x := 1$ as a valid slice. If the slice is being determines as part of a program analysis or comprehension task, then the programmer might (incorrectly) conclude that the original program assigns the value 1 to x whenever it terminates.

These considerations led to the development of the concept of a *semi-refinement*:

Definition 2.4 A *semi-refinement* of **S** is any program $\mathbf{S}'$ such that $\{\mathrm{WP}(\mathbf{S}, \textbf{true})\};\ \mathbf{S}'$ is equivalent to **S**. The semi-refinement relationship is denoted $\mathbf{S} \preccurlyeq \mathbf{S}'$.

The weakest precondition $\mathrm{WP}(\mathbf{S}, \textbf{true})$ is true on precisely those initial states for which **S** is guaranteed to terminate. Hence, the assertion $\{\mathrm{WP}(\mathbf{S}, \textbf{true})\}$ is a **skip** when **S** terminates and **abort** when **S** may not terminate.

If $\mathbf{S} \preccurlyeq \mathbf{S}'$ then $\mathbf{S}'$ must be equivalent to **S** when **S** terminates, but $\mathbf{S}'$ can do anything at all when **S** does not terminate. In particular $\mathbf{S}'$ can be equivalent to **skip** when **S** does not terminate.

We define a syntactic slice of **S** on X to be any reduction of **S** which is also a semi-refinement:

Definition 2.5 A *Syntactic Slice* of **S** on X is any program $\mathbf{S}'$ such that $\mathbf{S}' \sqsubseteq \mathbf{S}$ and

$$\mathbf{S};\ \textbf{remove}(W \setminus X) \preccurlyeq \mathbf{S}';\ \textbf{remove}(W \setminus X)$$

where W is the final state space for **S** and $\mathbf{S}'$.

We can extend this definition to slicing at arbitrary points in the program by adding assignments to a new slice variable as discussed in Section 1.

2.3. Semantic Slicing

The definition of a syntactic slice immediately suggests a generalisation: why restrict the semi-refinements to deleting statements? Or, to put it another way, why not drop the requirement that $\mathbf{S}' \sqsubseteq \mathbf{S}$?

Harman and Danicic [3,5] coined the term "amorphous program slicing" for a combination of slicing and transformation of executable programs. So far the transformations have been restricted to restructuring snd simplifications, but the definition of an amorphous slice allows any transformation (in any transformation theory) of executable programs.

We define a "semantic slice" to be any semi-refinement in WSL, so the concepts of semantic slicing and amorphous slicing are distinct but overlapping. A semantic slice is defined in the context of WSL transformation theory, while an amorphous slice is defined in terms of executable programs (WSL allows nonexecutable statements including abstract specification statements and guard statements). Also, amorphous slices are restricted to finite programs, while WSL programs (and hence, semantic slices) can include infinitary formulae. To summarise:

1. Amorphous slicing is restricted to finite, executable programs. Semantic slicing applies to any WSL programs including non-executable specification statements, non-executable guard statements, and programs containing infinitary formulae;
2. Semantic slicing is defined in the particular context of the WSL language and transformation theory: amorphous slicing applies to any transformation theory or definition of program equivalence on executable programs.

The relation between a WSL program and its semantic slice is a purely semantic one: compare this with a "syntactic slice" where the relation is primarily a syntactic one with a semantic restriction.

Definition 2.6 A *semantic slice* of $\mathbf{S}$ on X is any program $\mathbf{S}'$ such that:

$$\mathbf{S};\ \textsf{\textbf{remove}}(W \setminus X) \preccurlyeq \mathbf{S}';\ \textsf{\textbf{remove}}(W \setminus X)$$

Note that while there are only a finite number of different syntactic slices (if $\mathbf{S}$ contains n statements then there are at most 2^n different programs $\mathbf{S}'$ such that $\mathbf{S}' \sqsubseteq \mathbf{S}$) there are infinitely many possible semantic slices for a program: including slices which are actually *larger* than the original program. Although one would normally expect a semantic slice to be no larger than the original program, [11, 12] discuss cases where a high-level abstract specification can be larger than the program while still being arguably easier to understand and more useful for comprehension and debugging. A program might use some very clever coding to re-use the same data structure for more than one purpose. An equivalent program which internally uses two data structures might contain more statements and be less efficient while still being easier to analyse and understand.

2.4. Conditioned Slicing

A *conditioned slice* [4] is a slice of a program to which extra conditions have been added in the form of assertions. These conditions can allow further statements to be deleted. The formal definitions of conditioned syntactic slicing and conditioned semantic slicing are therefore identical to the definitions of syntactic and semantic slicing with suitable assertion statements inserted in the original program.

3. Minimal Semantic Slicing

Theorem 3.1 *The Representation Theorem*
Let $\mathbf{S}$ be any statement with initial state space V and final state space V. Let $\mathbf{x}$ be a list of all the variables in V. Then $\mathbf{S}$ is equivalent to:
$[\neg\mathrm{WP}(\mathbf{S}, \textsf{\textbf{false}})];$
$\mathbf{x} := \mathbf{x}'.(\neg\mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}') \ \wedge\ \mathrm{WP}(\mathbf{S}, \textsf{\textbf{true}}))$

If $\mathbf{S}$ is null-free (which is guaranteed for all WSL statements in language levels above the kernel level) then $\mathrm{WP}(\mathbf{S}, \textsf{\textbf{false}})$ is false, and the initial guard is redundant. For such statements we can transform the specification statement to show that $\mathbf{S}$ is equivalent to:
$\{\mathrm{WP}(\mathbf{S}, \textsf{\textbf{true}})\};\ \mathbf{x} := \mathbf{x}'.(\neg\mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}'))$

Then, by the definition of semi-refinement:

$$\mathbf{S} \preccurlyeq \mathbf{x} := \mathbf{x}'.(\neg\mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}'))$$

This is clearly a *minimal* semantic slice (counting statements) since it only contains a single statement, and by definition no WSL program can be smaller than a single statement. (It is not necessarily minimal if we are counting the total number of symbols: if statement $\mathbf{S}$ contains loops or recursion then the formula $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ is infinitely long!) So we have:

Theorem 3.2 *The Minimal Semantic Slice Theorem*
Let $\mathbf{S}$, be any null-free statement and let $\mathbf{x}$ be any list of variables. Then the statement $\mathbf{x} := \mathbf{x}'.(\neg\mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}'))$ is a valid semantic slice of $\mathbf{S}$ on the final values of $\mathbf{x}$.

This may appear to contradict Weiser's theorem on the non-computability of minimal slices, but Weiser's theorem only applies to algorithms for computing minimal *syntactic* slices. The construction of $\mathbf{x} := \mathbf{x}'.(\neg\mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}'))$ from $\mathbf{S}$, while being well defined, is not an algorithm in the usual sense because the formula WP$(\mathbf{S}, \mathbf{x} \neq \mathbf{x}')$ may be infinitely long. (In fact, it *will* be infinite whenever **S** contains any loops or recursion). An infinite specification statement is not directly executable, so this result is only practical for statements which contain no loops or recursion, but it does show that *no semantic slice need be larger than a single statement*.

For WSL programs with no loops or recursion (and where all the formulae are finite). Theorem 3.2 *does* give an algorithm for computing a minimal semantic slice on any given slicing criterion. By combining the representation theorem and a syntactic slicing algorithm with other program transformations we have developed a semantic slicer for general WSL programs.

4. Implementation of Abstraction and Refinement

We have implemented a function @WP in the FermaT transformation system which computes the weakest precondition for any program which does not include loops or procedure calls. (The implementation could be extended to non-recursive procedures and functions in the obvious way: by unfolding all procedures and functions in the main body of the program). With the aid of this function, we have implemented a transformation called Prog_To_Spec which can transform any non-recursive and non-iterative program into an equivalent specification statement. The implementation of @WP required less than 100 lines of $\mathcal{META}$WSL code, and the body of Prog_To_Spec is only 32 lines of code (including comments), demonstrating that $\mathcal{META}$WSL is the ideal language for implementing program transformations!

In the rest of the paper, all slicing examples were computed by FermaT in a single step and the output copied into the paper.

In ASCII notation, the WSL specification statement is written as follows:

```
SPEC <x1, x2, ... xn>: Q ENDSPEC
```

This statement assigns values to the variables x1, ..., xn such that the condition Q is satisfied. Within Q the *primed variables* x1', ..., xn' represent the new values to be assigned, and x1, ..., xn represent the original values. If there is no assignment of values which will satisfy Q, then the statement aborts. For example, the specification:

```
SPEC <x>: x' = x + 1 ENDSPEC
```

increments the value of x, while

```
SPEC <x>: EVEN(x) AND ODD(x') ENDSPEC
```

assigns x any odd value, provided it initially held an even value. (The first of the two specifications is a strict refinement of the second.)

Applying Prog_To_Spec to the assignment

```
x := 2 * x + 1
```

gives the specification:

```
SPEC <x>: x' = 2 * x + 1
```

Another example:

```
D_IF TRUE -> p := 1
  [] TRUE -> p := 2 FI
```

gives:

```
SPEC <p>: p' = 1 OR p' = 2 ENDSPEC
```

The statement to be specified may include assertions, local variables, nested IF statements and so on. FermaT's simplifier will use the assertions to simplify other parts of the generated specification, eliminate local variables and so on, automatically. For example:

```
VAR < x := y >:
IF p > q THEN x := x + 2
         ELSE x := x - 2 FI;
{x = 10} ENDVAR
```

is transformed to the assertion:

```
{y = 8 AND p > q OR y = 12 AND p <= q}
```

Note that the assertion has been used to simplify the preceding code.

A simple IF statement such as:

```
IF x > y THEN z := 1 ELSE z := 2 FI
```

is transformed to the specification:

```
SPEC <z>:
  z' = 1 AND x > y
    OR z' = 2 AND x <= y ENDSPEC
```

while a nested IF statement such as:

```
IF x = 1 THEN y := 2
ELSIF x = 2 THEN y := 3
            ELSE y := 4 FI
```

becomes:

```
SPEC <y>:
  y' = 4 AND x <> 1 AND x <> 2
    OR y' = 2 AND x = 1
    OR y' = 3 AND x = 2 ENDSPEC
```

Generally, programmers find that a compound statement with assertions, IF statements and simple assignments to be easier to read and understand than the equivalent single specification statement. So we have implemented another transformation Refine_Spec which analyses a specification statement and carries out the following operations:

1. Factor out any assertions;
2. Expand into an IF statement: for example, the specification $\mathbf{x} := \mathbf{x}'.(\mathbf{Q} \vee (\mathbf{B} \wedge \mathbf{P}))$ where **B** does not contain any variables $\mathbf{x}'$, is equivalent to
   if $\mathbf{B}$ **then** $\mathbf{x} := \mathbf{x}'.(\mathbf{Q}' \vee \mathbf{P}')$ **else** $\mathbf{x} := \mathbf{x}'.(\mathbf{Q}'')$ **fi**
   where $\mathbf{Q}'$ and $\mathbf{P}'$ are the result of simplifying **Q** and **P** under the assumption that **B** is true, and $\mathbf{Q}''$ is the result of simplifying **Q** under the assumption that **B** is false. These sub-specifications are then recursively refined;
3. Finally, any simple assignments or parallel assignments are extracted.

For example, the statement:

```
VAR < x := x >:
IF p = q
  THEN x := 18
  ELSE x := 17 FI;
IF p <> q
  THEN y := x
  ELSE  y := 2 FI ENDVAR
```

is abstracted to the specification:

```
SPEC <y>:
  y' = 2 AND p = q
    OR y' = 17 AND p <> q ENDSPEC
```

Applying Refine_Spec produces:

```
IF p = q THEN y := 2 ELSE y := 17 FI
```

The above example shows one way in which abstraction and refinement can be applied to construct a semantic slice: simply convert all the assigned variables that we do *not* want to slice on (x in this case) into local variables, and apply abstraction and refinement to the result!

5. Implementation of Conditioned Semantic Slicing

We have implemented a semantic slicer by combining abstraction and refinement (via the Prog_To_Spec and Refine_Spec transformations) with a simple syntactic slicing algorithm plus some additional general-purpose transformations. The slicer is also a *conditioned* slicer: since any assertions added to the program before slicing are used by the abstraction and refinement transformations to simplify the program.

The heart of the slicer is the @SSlice function which takes a WSL program and a set of variables (the slicing criterion for the end of the program) and returns a list of two elements: the sliced program and the set of variables whose values are required at the beginning of the program.

The main body of the semantic slice transformation executes @SSlice in a loop:

```
do $R :=$ @SSlice(@I, $X$);
   Make $R[1]$ the current program;
   Apply the transformation Refine_Spec
   to any specification statements;
   Apply Constant_Propagation;
   if the program has not changed in this iteration
     then exit(1) fi od
```

The @SSlice(I, X) function is outlined in Figure 1.

Deleting an assertion is a valid slice, though other strict refinements are not considered to be valid. However, our semantic slicer does not delete assertions since this might destroy the conditioning information before it can be applied to the program. If required, a call to Delete_Assertions can be added at the end of the transformation.

5.1. The Tax Calculation Program

The program in Figure 2 computes the amount of tax payable, including allowances, for a UK citizen in the tax year April 1998 to April 1999. Each person has a personal allowance, which is not taxed. This depends on their status, reflected in the variables married, blind, widowed, and the integer variable age. There are three tax bands, for which tax is charged at the rates 10%, 23% and 40%.

With FermaT's conditioned semantic slicer we can extract the business rules for particular situations and express them in a concise and readable format.

In [4] the conditioned program slicer ConSIT was used to compute a conditioned slice for the tax program (implemented in a subset of C) to answer the question "What is the personal allowance calculation for a blind widow aged over 68?". ConSIT produced the following program as the answer (which we have translated from C to WSL):

```
IF age >=75
  THEN personal := 5980
ELSE IF age >= 65
THEN personal := 5720 FI FI;
IF age >= 65 AND income > 16800
  THEN VAR < t := personal -
                    (income-16800)/2 >:
IF t > 4335
  THEN personal := t
  ELSE personal := 4335 FI ENDVAR FI;
IF blind = 1
  THEN personal := personal + 1380 FI
```

```
funct @SSlice$(I, X)$ $\equiv$
var $\langle R := \langle\rangle$, new $:= \langle\rangle$, newX $:= \langle\rangle\rangle$ :
  if $I$ is a statement sequence
    then Trim trailing statements from $I$ which do not
         assign to any variables in $X$ fi
  If there are no loops in $I$, and it is not a simple
  assertion or abort statement, then apply
  Prog_To_Spec to convert it to a specification;
  if $I$ is a statement sequence $\mathbf{S}_1;\ \ldots;\ \mathbf{S}_n$
    then for $S \in \langle \mathbf{S}_n, \ldots, \mathbf{S}_1 \rangle$ do
             $R :=$ @SSlice$(S, X)$;
             new $:= \langle R[1] \rangle +\!\!+$ new;
             $X := R[2]$ od;
         $R := \langle$fill Statements ~*new endfill$, X\rangle$
  elsif $I$ is an assertion or comment
      then $R := \langle I, X \rangle$
  elsif $I$ is an abort
      then $R := \langle I, \varnothing \rangle$
  elsif no variable assigned in $I$ is in $X$
      then $R := \langle$@Skip$, X\rangle$
  elsif $I$ is if $\mathbf{B}$ then $\mathbf{S}_1$ else $\mathbf{S}_2$ fi
      then var $\langle R_1 :=$ @SSlice$(\mathbf{S}_1, X)$,
                 $R_2 :=$ @SSlice$(\mathbf{S}_2, X)\rangle$ :
             $R := \langle$if $\mathbf{B}$ then $R_1[1]$ else $R_1[2]$ fi,
                  $R_1[2] \cup R_2[2] \cup$ vars$(\mathbf{B})\rangle$ end
  elsif $I$ is while $\mathbf{B}$ do $\mathbf{S}$ od
      then var $\langle B :=$ vars$(\mathbf{B})$, newX $:= X\rangle$ :
             do $R :=$ @SSlice$(S$, newX$)$;
                $R[2] := R[2] \cup$ newX $\cup B$;
                if $R[2] =$ newX then exit$(1)$ fi;
                newX $:= R[2]$ od end;
           $R := \langle$while $\mathbf{B}$ do $R[1]$ od$, R[2]\rangle$
  elsif $I$ is var $\langle v := e \rangle : \mathbf{S}$ end
      then $R :=$ @SSlice$(\mathbf{S}, X \setminus \{v\})$;
           newX $:= (R[2] \setminus \{v\}) \cup (\{v\} \cap X)$;
           if $v \in R[2]$
             then $R := \langle$var $\langle v := e \rangle : R[2]$ end, newX$\rangle$
             else $R := \langle$var $\langle v := \perp \rangle : R[2]$ end, newX$\rangle$ fi
  elsif $I$ is $\mathbf{x} := \mathbf{x}'.\mathbf{Q}$
      then var $\langle \mathbf{x}_1 := \mathbf{x} \cap X, \mathbf{x}_2 := \mathbf{x} \setminus X, R_1 := \langle\rangle\rangle$ :
             $R_1 := \mathbf{x}_1 := \mathbf{x}_1'.\exists \mathbf{x}_2.\mathbf{Q}$;
             $R := \langle R_1, (X \setminus$ vars$(\mathbf{x}_1)) \cup$ vars$(R_1)\rangle$ end.
```

Figure 1. Definition of @SSlice

```
IF age >= 75
  THEN personal := 5980
  ELSE IF age >= 65
         THEN personal := 5720
         ELSE personal := 4335 FI FI;
IF age >= 65 AND income > 16800
    THEN VAR < t := personal
                    - (income - 16800)/2 >:
         IF t > 4335
           THEN personal := t
           ELSE personal := 4335 FI
         ENDVAR FI;
IF blind = 1
  THEN personal := personal + 1380 FI;
IF married = 1 AND age >= 75
  THEN pc10 := 6692
  ELSE IF married = 1 AND age >= 65
         THEN pc10 := 6625
         ELSE IF married = 1 OR widow = 1
                THEN pc10 := 3470
                ELSE pc10 := 1500 FI FI FI;
IF married = 1 AND age >= 65
     AND income > 16800
  THEN VAR < t := pc10
                  - ((income - 16800)/2) >:
       IF t > 3740
         THEN pc10 := t
         ELSE pc10 := 3740 FI ENDVAR FI;
IF income <= personal
  THEN tax := 0
  ELSE
    income := income - personal;
    IF income <= pc10
      THEN tax := income * rate10
      ELSE
        tax := pc10 * rate10;
        income := income - pc10;
        IF income <= 28000
          THEN tax := tax + income * rate23
          ELSE tax := tax + 28000 * rate23;
               income := income - 28000;
               tax := tax + income * rate40
        FI FI FI
```

Figure 2. Tax Calculation Program

To carry out the same computation with FermaT's semantic slicer, we simply insert these assertions at the top of the program:

```
{blind = 1};
{married = 0};
{widow = 1};
{age > 68};
```

and then slice on the value of the variable `personal`. The resulting semantic slice is:

```
IF age < 75 AND income >= 19570
  THEN personal := 5715
ELSIF age < 75 AND income > 16800
  THEN personal := (16800 - income)/2 + 7100
ELSIF age < 75
  THEN personal := 7100
ELSIF income >= 20090
  THEN personal := 5715
ELSIF income > 16800
  THEN personal := (16800 - income)/2 + 7360
  ELSE personal := 7360 FI
```

Although both slices have the same number of lines of code, the semantic slice is clearly easier to understand: each input condition leads to a single assignment to `personal` which either gives the final value, or computes the value from `income`.

The tax calculation for a sighted, married person aged between 65 and 74 with an income over 30,000 can be computed by adding the assertions:

```
{age >= 65 AND age < 75};
{income > 30000};
{blind = 0};
{married = 1};
```

and slicing on the final value of `tax`. The result is:

```
IF income <= 36075
  THEN tax := (income - 8075) * rate23
         + 3740 * rate10
  ELSE tax := (income - 36075) * rate40
                + 3740 * rate10
                + 28000 * rate23 FI
```

Conditioned slicing works equally well with conditions at the *end* of the program which relate to values calculated by the program. For example, if we want to determine which people with an income of less than 16,000 will have a personal allowance of 7,100 then we simply insert the assertion `{income < 16000}` at the top of the program, and insert `{personal = 7100}` at the end of the program and slice on the final value of `personal`. The result is:

```
{blind = 1
   AND age < 75
   AND income < 16000
   AND age >= 65};
personal := 7100
```

i.e. only blind people aged 65 to 74 will have a personal allowance of 7,100.

The following program is another example of using an assertion on the output of a program to simplify the code before the assertion:

```
IF x = 0 THEN y := 1
ELSIF x = 1 THEN y := 10
ELSIF x = 2 THEN y := 4
            ELSE y := 5 FI;
IF z = 1 THEN y := y + 1 FI;
{y < 5}
```

Slicing on `y` gives:

```
IF z = 1
  THEN {x = 0}; y := 2
ELSIF x = 2
  THEN y := 4
  ELSE {x = 0}; y := 1 FI
```

To slice on the condition that a certain branch in the program will not be taken, it is sufficient to insert an **abort** statement (which is equivalent to the assertion {**false**}) at the appropriate point or points. Many large commercial systems contain a lot of error handling code: in some cases much of the code in a module is for error handling and this can obscure the algorithms computed by the module. In addition, many modules produce more than one output. By inserting **abort** statements at all the points where an error has been detected, and slicing on each individual output it is possible to compute a concise representation of the algorithm (the "business rule") for each output of the module under normal conditions.

5.2. Semantic Slicing on Loops

The following program was used as an example in [2] and [4] which we have translated from C to WSL:

```
i := 1;
posprod := 1; negprod := 1;
possum := 0; negsum := 0;
WHILE i <= n DO
  a := input[i];
  {a > 0};
  IF a > 0
    THEN possum := possum + a;
         posprod := posprod * a
  ELSIF a < 0
    THEN negsum := negsum - a;
         negprod := negprod * (-a)
  ELSIF test0 = 1
    THEN IF possum >= negsum
           THEN possum := 0;
           ELSE negsum := 0 FI;
         IF posprod >= negprod
           THEN posprod := 1
           ELSE negprod := 1 FI FI;
  i := i + 1 OD;
IF possum >= negsum
  THEN sum := possum
  ELSE sum := negsum FI;
IF posprod >= negprod
  THEN prod := posprod
  ELSE prod := negprod FI
```

ConSIT took about 20 minutes on a Pentium II running at 233MHz to compute the following conditioned slice for the final value of sum:

```
i := 1;
possum := 0;
negsum := 0;
WHILE i <= n DO
  a := input[i];
  {a > 0};
  IF a > 0
    THEN possum := possum + a FI;
  i := i + 1 OD;
IF possum >= negsum
  THEN sum := possum FI
```

Note that ConSIT was unable to remove the variable negsum since it is a purely syntactic conditioned slicer. FermaT's semantic slicer took just over half a second (0.56 seconds) on a 3GHz PC to produce the slice:

```
i := 1;
possum := 0;
WHILE i <= n DO
  {input[i] > 0};
  < possum := input[i] + possum,
    i := i + 1 > OD;
IF possum < 0
  THEN sum := 0
  ELSE sum := possum FI
```

Here, the initial slice over the program deleted the assignments to negsum in the loop body. A subsequent Constant_Propagation replaced references to negsum by zero, after which a second slicing operation removed negsum from the program. All these transformations were carried out automatically by Semantic_Slice to give the result above.

5.3. Performance

The worst case performance of Semantic_Slice is exponential in the size of the program, as can be seen in the following simple sequence of n assignments:

```
x := f(x, x);
x := f(x, x);
...
x := f(x, x);
```

Any semantic or amorphous slicer which merges sequences of assignments to the same variable will give a single assignment which contains 2^n references to x.

In addition, a sequence of simple IF statements can also lead to exponential growth in the output since the weakest precondition for **if B then** $\mathbf{S}_1$ **else** $\mathbf{S}_2$ **fi** on postcondition **R** is:

$$(\mathbf{B} \Rightarrow \mathrm{WP}(\mathbf{S}_1, \mathbf{R})) \ \wedge \ (\neg\mathbf{B} \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{R}))$$

which contains two copies of **R**. The weakest precondition for two consecutive IF statements will therefore contain four copies of **R**, and so on. If the resulting conditions cannot be further simplified, then the abstracted specification statement may be exponentially large compared to the size of the input.

In most practical programs this exponential growth is unlikely to occur, so a practical solution is to limit the expansion of statements to ensure that the computed slices are no larger than the input program.

6. Future Work

There are many other transformations implemented in FermaT which could usefully be applied to semantic slicing: in particular, the restructuring and simplifying transformations. A transformation called Use_Assertion uses a given assertion to simplify subsequent code: this would make the processing of the tax examples much more efficient (we did not make use of Use_Assertion in this paper because we wanted to show that more powerful results can be achieved via abstraction and refinement. But Use_Assertion is able to propagate an assertion across a loop).

Another useful extension would be to attempt to derive and prove the correctness of simple invariants over loops in the program.

Transformations which collapse simple loops to MAP or REDUCE statements will allow the abstraction transformation Prog_To_Spec to be applied to larger blocks of code For example, the **while** loop in the sliced program in Section 5.2 can be transformed to an assignment of a REDUCE:

```
i := 1;
{REDUCE("+", input[1..n]) > 0};
possum := REDUCE("+", input[1..n]);
i := n + 1;
IF possum < 0
  THEN sum := 0
  ELSE sum := possum FI
```

Semantic_Slice on sum reduces this to the simple assignment:

```
sum := REDUCE("+", input[1..n])
```

FermaT includes a more powerful syntactic slicer which can process unstructured programs and procedures. Integrating abstraction and refinement into this slicer would produce a semantic conditioned slicer which can handle more WSL syntax.

7. Conclusion

In this paper we have described a partial implementation of the representation theorem in the FermaT transformation system. This enables the "abstraction" of any non-iterative and non-recursive WSL statement to generate an equivalent specification statement. The FermaT expression and condition simplifier can be applied to simplify the specification, which can then, if necessary, be refined into IF statements, assertions and simple assignments. This abstraction and refinement process has been integrated into a simple syntactic slicer, together with other FermaT transformations, to produce a powerful conditioned semantic slicer (which is *not* restricted to non-iterative programs!) The slicer is able to use both preconditions and postconditions to simplify the code before and after slicing

FermaT is available under the GNU GPL (General Public Licence) from the following web sites:

http://www.dur.ac.uk/∼martin.ward/fermat.html
http://www.cse.dmu.ac.uk/∼mward/fermat.html

8. References

[1] David W. Binkley & Keith Gallagher, "A survey of program slicing," in *Advances in Computers*, Marvin Zelkowitz, ed., Academic Press, San Diego, CA, 1996.

[2] G. Canfora, A. Cimitile & A. De Lucia, "Conditioned program slicing," *Information and Software Technology Special Issue on Program Slicing* 40 (1998), 595–607.

[3] Mark Harman & Sebastian Danicic, "Amorphous program slicing," *5th IEEE International Workshop on Program Comprehesion (IWPC'97), Dearborn, Michigan, USA* (May 1997).

[4] Mark Harman, Sebastian Danicic & R. M. Hierons, "ConSIT: A conditioned program slicer," *9th IEEE International Conference on Software Maintenance (ICSM'00), San Jose, California, USA*, Los Alamitos, California, USA (Oct., 2000).

[5] Mark Harman, Lin Hu, Malcolm Munro & Xingyuan Zhang, "GUSTT: An Amorphous Slicing System Which Combines Slicing and Transformation," *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, Los Alamitos, California, USA (2001).

[6] J. C. Lagarias, "The $3x + 1$ Problem and Its Generalizations," *American Mathematical Monthly* 92 (1985), 3–23, ⟨http://www.cecm.sfu.ca/organics/papers/lagarias/⟩.

[7] B. Thwaites, "Two Conjectures, or How to Win £1100," *Mathematical Gazette* 80 (1996), 35–36.

[8] F. Tip, "A Survey of Program Slicing Techniques," *Journal of Programming Languages,* 3 (Sept., 1995), 121–189.

[9] M. Ward, "Proving Program Refi nements and Transformations," Oxford University, DPhil Thesis, 1989.

[10] M. Ward, "The Formal Transformation Approach to Source Code Analysis and Manipulation," *IEEE International Workshop on Source Code Analysis and Manipulation Florence, Italy, 10th November*, Los Alamitos, California, USA (2001).

[11] M. Ward, "Abstracting a Specifi cation from Code," *J. Software Maintenance: Research and Practice* 5 (June, 1993), 101–122, ⟨http://www.dur.ac.uk/martin.ward/martin/papers/prog-spec.ps.gz⟩.

[12] M. Ward, "A Defi nition of Abstraction," *J. Software Maintenance: Research and Practice* 7 (Nov., 1995), 443–450, ⟨http://www.dur.ac.uk/martin.ward/martin/papers/abstraction-t.ps.gz⟩.

[13] M. Ward, "Derivation of Data Intensive Algorithms by Formal Transformation," *IEEE Trans. Software Eng.* 22 (Sept., 1996), 665–686, ⟨http://www.dur.ac.uk/martin.ward/martin/papers/sw-alg.ps.gz⟩.

[14] M. Weiser, "Program slicing," *IEEE Trans. Software Eng.* 10 (July, 1984), 352–357.

[15] M. Weiser, "Programmers use slices when debugging," *Comm. ACM* 25 (July, 1984), 352–357.