Program Slicing via FermaT Transformations

M. P. Ward
Software Technology Research Lab
De Montfort University
The Gateway,
Leicester LE1 9BH, UK
Martin.Ward@durham.ac.uk

Abstract

In this paper we give a brief introduction to the foundations of WSL transformation theory and describe how the concept of program slicing can be formalised in the theory. This formalism naturally lends itself to several generalisations including amorphous slicing and conditioned slicing. One novel generalisation is "semantic slicing" which combines slicing and abstraction to a specification. Interprocedural semantic slicing has been implemented in the FermaT transformation system [16]: an industrial-strength transformation system designed for forward and reverse engineering, re-engineering and program comprehension.

1 Introduction

In the development of methods for program analysis and manipulation it is important to start from a rigorous mathematical foundation. Without such a foundation, it is all too easy to assume that a particular transformation is valid, and come to rely upon it, only to discover that there are certain special cases where the transformation is not valid.

The way to get a rigorous proof of the correctness of a transformation is to first define precisely when two programs are "equivalent", and then show that the transformation in question will turn any suitable program into an equivalent program. To do this, we need to make some simplifying assumptions: for example, we usually ignore the execution time of the program. This is not because we do not care about efficiency—far from it—but because we want to be able to use the theory to prove the correctness of optimising transformations: where a program is transformed into a more efficient version.

More generally, we ignore the internal sequence of state changes that a program carries out: we are only interested in the initial and final states (but see Section 4 for a discussion of operational semantics).

Our mathematical model defines the semantics of a program as a function from states to sets of states. For each initial state s, the function f returns the set of states $f(s)$ which are all the possible final states of the program when it is started in state s. A special state $\perp$ indicates nontermination or an error condition. If $\perp$ is in the set of final states, then the program might not terminate for that initial state (in which case, we put all the other states into $f(s)$).

If two programs have the same semantic function then they are *equivalent*. A *transformation* is an operation which takes any program satisfying its applicability conditions and returns an equivalent program. A *refinement* of a program is any program which terminates on all the initial states for which the original program terminates, and for each such state it is guaranteed to terminate in a possible final state for the original program. In terms of semantic functions, the relation is:

$$\forall s.\, f_2(s) \subseteq f_1(s)$$

since if f_1 may not terminate on s then $f_1(s)$ contains all possible states, including $\perp$.

2 Transformation Theory

Our transformation theory developed in roughly the following stages:

1. Start with a very simple and tractable kernel language;
2. Develop proof techniques based on set theory and mathematical logic, for proving the correctness of transformations in the kernel language;

3. Extend the kernel language by definitional transformations which introduce new constructs (the result is the WSL wide spectrum language);
4. Develop a catalogue of proven WSL transformations: each transformation is proved correct by appealing to already proven transformations, or by translating to the kernel language and applying the proof techniques directly.
5. Tackle some challenging program development and reverse engineering tasks to demonstrate the validity of this approach;
6. Extend WSL with constructs for implementing program transformations (the result is called $\mathcal{METAWSL}$);
7. Implement an industrial strength transformation engine in $\mathcal{META}$WSL with translators to and from existing programming languages. This allowed us to test our theories on large scale legacy systems (including systems written in IBM Assembler: see [16,17,20]).

2.1 The Kernel Language

It turns out that for our kernel language we can do away with many familiar programming constructs: including assignments and **if** statements. We need just four primitive statements and three compound statements. Let **P** and **Q** be any logical formulae (technically, these are formulae of an infinitary first order logic) and **x** and **y** be any finite lists of variables. The primitive statements are:

1. **Assertion:** $\{\mathbf{P}\}$ is an assertion statement which acts as a partial **skip** statement. If the formula **P** is true then the statement terminates immediately without changing any variables, otherwise it aborts (we treat abnormal termination and non-termination as equivalent, so a program which aborts is equivalent to one which never terminates);
2. **Guard:** $[\mathbf{Q}]$ is a guard statement. It always terminates, and enforces **Q** to be true at this point in the program *without changing the values of any variables.* It has the effect of restricting previous nondeterminism to those cases which will cause **Q** to be true at this point. If this cannot be ensured then the set of possible final states is empty, and therefore all the final states will satisfy any desired condition (including **Q**);
3. **Add variables:** **add**(**x**) adds the variables in **x** to the state space (if they are not already present) and assigns arbitrary values to them. The arbitrary values may be restricted to particular values by a subsequent guard;
4. **Remove variables:** **remove**(**y**) removes the variables in **y** from the state space (if they are present).

while the compound statements are:

1. **Sequence:** $(\mathbf{S}_1;\ \mathbf{S}_2)$ executes $\mathbf{S}_1$ followed by $\mathbf{S}_2$;
2. **Nondeterministic choice:** $(\mathbf{S}_1 \sqcap \mathbf{S}_2)$ chooses one of $\mathbf{S}_1$ or $\mathbf{S}_2$ for execution, the choice being made nondeterministically;
3. **Recursion:** $(\mu X.\mathbf{S}_1)$ where X is a *statement variable* (a symbol taken from a suitable set of symbols). The statement $\mathbf{S}_1$ may contain occurrences of X as one or more of its component statements. These represent recursive calls to the procedure whose body is $\mathbf{S}_1$.

Some of these constructs, particularly the guard statement, may be unfamiliar to many programmers, while other more familiar constructs such as assignments and conditional statements appear to be missing. It turns out that assignments and conditionals, which used to be thought of as "atomic" operations, can be constructed out of these more fundamental constructs. On the other hand, the guard statement by itself is unimplementable in any programming language: for example, the guard statement [**false**] is guaranteed to terminate in a state in which **false** is true. In the semantic model this is easy to achieve: the semantic function for [**false**] has an *empty* set of final states for each proper initial state. As a result, [**false**] is a valid refinement for *any* program. Morgan [11] calls this construct "miracle". Such considerations have led to the Kernel language constructs being described as "the Quarks of Programming": mysterious entities which cannot be observed in isolation, but which combine to form what were previously thought of as the fundamental particles.

Assignments can be constructed from a sequence of **add**statements and guards. For example, the assignment $x := 1$ is constructed by adding x and restricting its value: $(\mathbf{add}(\langle x \rangle);\ [x = 1])$. For an assignment such as $x := x + 1$ we need to record the new value of x in a new variable, x' say, before copying it into x. So we can construct $x := x+1$ as follows: $(\mathbf{add}(\langle x' \rangle);\ ([x' = x + 1];\ (\mathbf{add}(\langle x \rangle);\ ([x = x'];\ \mathbf{remove}(x')))))$.

Conditional statements are constructed by combining guards with nondeterministic choice. For example, **if B then** $\mathbf{S}_1$ **else** $\mathbf{S}_2$ **fi** can be constructed as $(([\mathbf{B}];\ \mathbf{S}_1) \sqcap ([\neg\mathbf{B}];\ \mathbf{S}_2))$.

2.2 The Specification Statement

For our transformation theory to be useful for both forward and reverse engineering it is important to be able to represent abstract specifications as part of the language. Then the refinement of a specification into an executable program, or the reverse process of abstracting a specification from executable code, can both be carried out within a single language. We define the notation $\mathbf{x} := \mathbf{x}'.\mathbf{Q}$ where $\mathbf{x}$ is a sequence of variables and $\mathbf{x}'$ the corresponding sequence of "primed variables", and $\mathbf{Q}$ is any formula. This assigns new values to the variables in $\mathbf{x}$ so that the formula $\mathbf{Q}$ is true where (within $\mathbf{Q}$) $\mathbf{x}$ represents the old values and $\mathbf{x}'$ represents the new values. If there are no new values for $\mathbf{x}$ which satisfy $\mathbf{Q}$ then the statement aborts. The formal definition is:

$$\mathbf{x} := \mathbf{x}'.\mathbf{Q} \ =_{\text{DF}} (\{\exists \mathbf{x}'.\,\mathbf{Q}\};\ (\textsf{add}(\mathbf{x}');\ ([\mathbf{Q}];\ (\textsf{add}(\mathbf{x});\ ([\mathbf{x} = \mathbf{x}'];\ \textsf{remove}(\mathbf{x}'))))))$$

2.3 Semantic Refinement

A *state* is a collection of variables (the *state space*) with values assigned to them; thus a state is a function which maps from a finite set V of variables to a set $\mathcal{H}$ of values. There is a special extra state $\perp$ which is used to represent nontermination or error conditions. (It does not give values to any variables). States other than $\perp$ are called *proper states*. A state transformation f maps each initial state s in one state space, to the set of possible final states $f(s)$, which may be in a different state space. If $\perp$ is in $f(s)$ then, by definition, so is every other state, also $f(\perp)$ is the set of all states (including $\perp$).

Semantic refinement is defined in terms of these state transformations. A state transformation f is a refinement of a state transformation g if they have the same initial and final state spaces and $f(s) \subseteq g(s)$ for every initial state s. Note that if $\perp \in g(s)$ for some s, then by definition $g(s)$ includes every state, so $f(s)$ can be anything at all. In other words we can correctly refine an "undefined" program to do anything we please. If f is a refinement of g (equivalently, g is refined by f) we write $g \ \leq\ f$.

A *structure* for a logical language $\mathcal{L}$ consists of a set of values, plus a mapping between constant symbols, function symbols and relation symbols of $\mathcal{L}$ and elements, functions and relations on the set of values. If the interpretation of statement $\mathbf{S}_1$ under the structure M is refined by the interpretation of statement $\mathbf{S}_2$ under the same structure, then we write $\mathbf{S}_1 \leq_M \mathbf{S}_2$. A *model* for a set of sentences (formulae with no free variables) is a structure for the language such that each of the sentences is interpreted as true. If $\mathbf{S}_1 \leq_M \mathbf{S}_2$ for every model M of a countable set Δ of sentences of $\mathcal{L}$ then we write $\Delta \models \mathbf{S}_1 \ \leq\ \mathbf{S}_2$.

Here Δ is the set of assumptions about the logical symbols under which the refinement is valid.

2.4 Weakest Preconditions

Given any statement $\mathbf{S}$ and any formula $\mathbf{R}$ which defines a condition on the final states for $\mathbf{S}$, we define the *weakest precondition* $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ to be the weakest condition on the initial states for $\mathbf{S}$ such that if $\mathbf{S}$ is started in any state which satisfies $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ then it is guaranteed to terminate in a state which satisfies $\mathbf{R}$. By using an infinitary logic, it turns out that $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ has a simple definition for all kernel language programs $\mathbf{S}$ and all (infinitary logic) formulae $\mathbf{R}$:

$$\begin{aligned}
\mathrm{WP}(\{\mathbf{P}\}, \mathbf{R}) &=_{\text{DF}} \mathbf{P} \wedge \mathbf{R} \\
\mathrm{WP}([\mathbf{Q}], \mathbf{R}) &=_{\text{DF}} \mathbf{Q} \Rightarrow \mathbf{R} \\
\mathrm{WP}(\textsf{add}(\mathbf{x}), \mathbf{R}) &=_{\text{DF}} \forall \mathbf{x}.\,\mathbf{R} \\
\mathrm{WP}(\textsf{remove}(\mathbf{x}), \mathbf{R}) &=_{\text{DF}} \mathbf{R} \\
\mathrm{WP}((\mathbf{S}_1;\ \mathbf{S}_2), \mathbf{R}) &=_{\text{DF}} \mathrm{WP}(\mathbf{S}_1, \mathrm{WP}(\mathbf{S}_2, \mathbf{R})) \\
\mathrm{WP}((\mathbf{S}_1 \sqcap \mathbf{S}_2), \mathbf{R}) &=_{\text{DF}} \mathrm{WP}(\mathbf{S}_1, \mathbf{R}) \wedge \mathrm{WP}(\mathbf{S}_2, \mathbf{R}) \\
\mathrm{WP}((\mu X.\mathbf{S}), \mathbf{R}) &=_{\text{DF}} \bigvee_{n<\omega} \mathrm{WP}((\mu X.\mathbf{S})^n, \mathbf{R})
\end{aligned}$$

where $(\mu X.\mathbf{S})^0 = \textsf{abort} = \{\textsf{false}\}$ and $(\mu X.\mathbf{S})^{n+1} = \mathbf{S}[(\mu X.\mathbf{S})^n/X]$ which is $\mathbf{S}$ with all occurrences of X replaced by $(\mu X.\mathbf{S})^n$. (In general, for statements $\mathbf{S}$, $\mathbf{T}$ and $\mathbf{T}'$, the notation $\mathbf{S}[\mathbf{T}'/\mathbf{T}]$ means $\mathbf{S}$ with $\mathbf{T}'$ instead of each $\mathbf{T}$).

3 Extensions to the Kernel Language

The WSL language is built up in a set of layers, starting with the kernel language. The first level language includes specification statements, assignments, **if** statements, **while** and **for** loops, Dijkstra's guarded commands [7] and simple local variables.

The second level language adds **do** ... **od** loops, action systems and true local variables.

A **do** ... **od** loop is an unbounded or "infinite" loop which can only be terminated by executing an **exit**(n) statement (where the n must be a simple integer, not a variable or expression). This statement terminates the n enclosing **do** ... **od** loop.

A *proper sequence* is any program such that every **exit**(n) is contained within at least n enclosing loops. If $\mathbf{S}$ contains an **exit**(n) within less than n loops, then this **exit** could cause termination of a loop enclosing $\mathbf{S}$. The set of terminal values of $\mathbf{S}$, denoted $\mathsf{TVs}(\mathbf{S})$ is the set of integers $n - d \geqslant 0$ such that there is an **exit**(n) within d nested loops in $\mathbf{S}$ which could cause termination of $\mathbf{S}$. $\mathsf{TVs}(\mathbf{S})$ also contains 0 if $\mathbf{S}$ could terminate normally (i.e. not via an **exit** statement).

For example, if $\mathbf{S}$ is the program:

```
do if $x = 0$ then exit(3)
   elsif $x = 1$ then exit(2) fi;
   $x := x - 2$ od
```

Then $\mathsf{TVs}(\mathbf{S}) = \{1, 2\}$.

We earlier mentioned the remarkable properties of the guard statement: in particular [**false**] is a valid refinement for any program or specification. This is because the set of final states is empty for every (proper) initial state. A program which may have an empty set of final states is called a *null program* and is inherently unimplementable in any programming language. So it is important to avoid inadvertantly introducing a null program as the result of a refinement process. Morgan [11] calls the program [**false**] a "miracle", after Dijkstra's "Law of Excluded Miracles" [7]:

$$\mathrm{WP}(\mathbf{S}, \mathsf{false}) = \mathsf{false}$$

Part of the motivation for our specification statement is to exclude null programs (Morgan leaves it to the programmer to ensure that null programs are not introduced by accident). Fortunately, any WSL program with no explicit guard statements is non-null and obeys Dijkstra's law.

4 Operational Semantics

The correctness proofs of WSL transformations only look at the external behaviour of the programs. To prove that a transformation also preserves the actual sequence of internal operations then it would appear that a new definition of the semantics of programs is required: one which defines the meaning of a program to be a function from the initial state to the possible sequences of internal states culminating in the final state of the program, in other words, an *operational semantics*. We would then need to attempt to re-prove the correctness of all the transformations under the new semantics, in order to find out which ones are still valid. But we would not have the benefit of the weakest precondition approach, and we would not be able to re-use any existing proofs.

It turns out that this extra work is not necessary: instead the operational semantics can be "encoded" in the denotational semantics. We add a new variable, seq, to the program which will be used to record the sequence of state changes. We then annotate the original program, adding assignments to seq at the appropriate places:

$$\begin{aligned} &\mathcal{A}(\textsf{if } \mathbf{B} \textsf{ then } \mathbf{S}_1 \textsf{ else } \mathbf{S}_2 \textsf{ fi}) \\ &\qquad =_{\text{DF}} \textsf{if } \mathbf{B} \textsf{ then } \mathcal{A}(\mathbf{S}_1) \textsf{ else } \mathcal{A}(\mathbf{S}_2) \textsf{ fi} \\ &\mathcal{A}(\mathbf{S}_1;\ \mathbf{S}_2) =_{\text{DF}} \mathcal{A}(\mathbf{S}_1);\ \mathcal{A}(\mathbf{S}_2) \\ &\mathcal{A}(v := e) =_{\text{DF}} \mathsf{seq} := \mathsf{seq} \mathbin{+\!\!+} \langle\langle \text{“v”}, e\rangle\rangle; \end{aligned}$$

and so on for the other constructs.

Given a transformation which turns $\mathbf{S}_1$ into the equivalent program $\mathbf{S}_2$, if we want to show that the transformation also preserves operational semantics it is sufficient to show that it turns the annotated program $\mathcal{A}(\mathbf{S}_1)$ into a program equivalent to $\mathcal{A}(\mathbf{S}_2)$.

5 Slicing

The notion of a program slice, originally introduced by Mark Weiser [21], has been found to be useful in program analysis, debugging and other areas. In Weiser's original definition, a slice of a program is taken with respect to a program point p and a variable x; the slice contains all statements of the program that might affect the value of x at point p. To be more precise, Weiser defined a program slice $\mathbf{S}$ as a *reduced, executable program* obtained from a program $\mathbf{P}$ by removing statements, such that $\mathbf{S}$ replicates part of the behavior of $\mathbf{P}$.

To give a formal definition of slicing in WSL we need to define a *reduction* of a program. We define the relation $\sqsubseteq$ on WSL programs as follows:

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

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

If $n > 0$ is the largest integer in $\mathsf{TVs}(\mathbf{S})$ then:

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

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

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

$$\begin{aligned} \textsf{while } \mathbf{B} \textsf{ do } \mathbf{S}' \textsf{ od} &\sqsubseteq \textsf{while } \mathbf{B} \textsf{ do } \mathbf{S} \textsf{ od} \\ \textsf{var } \langle v := e\rangle : \mathbf{S}' \textsf{ end} &\sqsubseteq \textsf{var } \langle v := e\rangle : \mathbf{S} \textsf{ end} \\ \textsf{var } \langle v := e\rangle : \mathbf{S}' \textsf{ end} &\sqsubseteq \textsf{var } \langle v := \bot\rangle : \mathbf{S} \textsf{ end} \end{aligned}$$

This last case will be used when the variable v is used in $\mathbf{S}$, but the initial value e is not used.

For a sequence of statements $\mathbf{S}_1;\ \mathbf{S}_2;\ \ldots;\ \mathbf{S}_n$ the reduction relation allows for deleting some of the statements and reducing the remainder. So a reduction of $\mathbf{S}$ is any sequence $\mathbf{S}'_1;\ \mathbf{S}'_2;\ \ldots;\ \mathbf{S}'_m$ where $m \leqslant n$ and there exists a monotonic, strictly increasing function $\phi : \{1,\ldots,m\} \rightarrow \{1,\ldots,n\}$ such that $\forall i, 1 \leqslant i \leqslant n.\, \mathbf{S}'_i \sqsubseteq \mathbf{S}_{phi(i)}$. In this case, we have:

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

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

5.1 Syntactic Slices

Initially we will consider the special case where p is the end point of the program, but we will generalise the variable x to a set X of variables. If X does not contain all the variables in the final state space of the program, then the sliced program will *not* be equivalent to the original program. However, consider the set $W \setminus X$, where W is the final state space. These are the variables whose values we are *not* interested in. By removing these variables from the final state space we can get a program which is equivalent to the sliced program. Suppose program $\mathbf{S}$ maps state space V to W (we write this as $\mathbf{S} : V \rightarrow W$), then the effect of slicing $\mathbf{S}$ at its end point on the variables in X is to generate a program equivalent to $\mathbf{S};\ \textsf{remove}(W \setminus X)$.

This suggests that we can define a slice of $\mathbf{S}$ on X to be any program $\mathbf{S}' \sqsubseteq \mathbf{S}$, such that:

$$\Delta \vdash \mathbf{S}';\ \textsf{remove}(W \setminus X)\ \approx\ \mathbf{S};\ \textsf{remove}(W \setminus X)$$

However, the requirement that the slice be strictly equivalent to the original program is too strict in some cases. Consider the program:

$$\mathbf{S};\ x := 0$$

where $\mathbf{S}$ does not contain any assignments to x. If we are slicing on x then we would like to delete the whole of $\mathbf{S}$: but the program $x := 0;\ \textsf{remove}(W \setminus \{x\})$ is only equivalent to $\mathbf{S};\ x := 0;\ \textsf{remove}(W \setminus \{x\})$ provided that $\mathbf{S}$ always terminates. But most slicing researchers see no difficulty in slicing away non-terminating code. One solution is to add an assertion to show that program equivalence is only required where the original program terminates. So we could define a slice of $\mathbf{S}$ on X to be any program $\mathbf{S}' \sqsubseteq \mathbf{S}$, such that:

$$\begin{aligned}\Delta \vdash \mathbf{S};\ &\textsf{remove}(W \setminus X)\\ &\approx\ \{\mathrm{WP}(\mathbf{S}, \textsf{true})\};\ \mathbf{S}';\ \textsf{remove}(W \setminus X)\end{aligned}$$

However, a simpler solution is to simply allow a slice to be a refinement of the original program. This would also allow us to slice nondeterministic programs by reducing the nondeterminism. For example we can extend the $\sqsubseteq$ operator so that:

$$\textsf{if true}\ \rightarrow\ \mathbf{S}_1\ \Box\ \textsf{true}\ \rightarrow\ \mathbf{S}_2\ \textsf{fi}$$

can be sliced to $\mathbf{S}_1$ or $\mathbf{S}_2$ as required.

Even without extending the $\sqsubseteq$ operator, simple deletion of statements can still create a refinement of the original program. For example, let $\mathbf{S}$ be the program:

```
$x := 0;$
$x := 1;$
if $x = 0 \vee x = 1 \rightarrow y := 1$
$\Box\ x = 1 \rightarrow y := 2$ fi;
$x := 0;$
```

This program assigns y the value 1 or 2 (the assignment is chosen nondeterministically) and x the value 0. If we delete the assignment $x := 1$ and the second $x := 0$ assignment, then the result is a deterministic refinement of $\mathbf{S}$ which assigns y the value 1. If strict refinement is not allowed then the best slice we can get for this program is to delete the first $x := 0$ assignment.

The foregoing discussion motivates this definition:

Definition 1 A *traditional slice* of $\mathbf{S}$ on X is any program $\mathbf{S} \sqsubseteq \mathbf{S}'$, such that:

$$\Delta \vdash \mathbf{S};\ \textsf{remove}(W \setminus X)\ \leq\ \mathbf{S}';\ \textsf{remove}(W \setminus X)$$

Within this framework, a proof of correctness of an algorithm for program slicing (such as the algorithm for interprocedural slicing in [10]) is simply a proof of the validity of the transformation which deletes the statements in $\mathbf{S}$ to create $\mathbf{S}'$.

5.2 Minimal Slices

Both Weiser's definition and ours allow the whole program as a valid slice for *any* slicing criterion, however restrictive that criterion is in comparison to the final state space. For program understanding and debugging, small slices are more useful than large slices, so it would appear to be a reasonable requirement to place on any slicing algorithm that the slices generated by the algorithm should be *minimal:* either in the sense of minimising the total number of statements, or at least in the weaker sense that no further statements can be deleted. For example, we define:

Definition 2 A *minimal slice* of $\mathbf{S}$ on X is any traditional slice $\mathbf{S}'$ such that if $\mathbf{S}'' \sqsubseteq \mathbf{S}'$ is also a traditional slice, then $\mathbf{S}'' = \mathbf{S}'$.

Note that a minimal slice, according to this definition, is not necessarily unique and is not necessarily a slice with the smallest number of statements. Consider the program $\mathbf{S}$:

$$x := 2;\ x := x + 1;\ x := 3$$

A traditional slice can be obtained from $\mathbf{S}$ by deleting the last statement to give $\mathbf{S}'$:

$$x := 2;\ x := x + 1$$

This program is a minimal slice (according to our definition), since neither of the remaining statements can be deleted. But there is another minimal slice of $\mathbf{S}$, namely $x := 3$, which has fewer statements than $\mathbf{S}'$.

Although of theoretical interest, the requirement that the slices be minimal is too restrictive to place on a slicing algorithm. This is because the general problem of finding a minimal slice is non-computable! Let $\mathbf{S}$ be any program and consider the program $\mathbf{S}'$ which is $\mathbf{S};\ x := 0$, where x is any variable which does not appear in $\mathbf{S}$. If $\mathbf{S}$ never terminates, then for any valid traditional slice of $\mathbf{S}'$ on $\{x\}$, if the slice still contains the assignment $x := 0$ then that statement can be deleted and the result will still be a valid slice of $\mathbf{S}'$. On the other hand, if $\mathbf{S}$ could terminate, then the sliced program has to set x to zero, so the final assignment must appear in any valid slice of $\mathbf{S}'$ on $\{x\}$. So if we had a program which computes minimal traditional slices, then we could solve the halting problem for any program $\mathbf{S}$ by computing the minimal slice of the program $\mathbf{S};\ x := 0$ on $\{x\}$ and simply observing if the result ends in the statement $x := 0$. If it does, then $\mathbf{S}$ terminates, while if it doesn't then $\mathbf{S}$ does not terminate.

5.3 Semantic Slice

The definition of a traditional slice immediately suggests a generalisation: why restrict the refinements to deleting statements? Or, to put it another way, why insist on the requirement that $\mathbf{S}' \sqsubseteq \mathbf{S}$? Harman and Danicic [8] coined the term "amorphous program slicing" for a combination of slicing and transformation, but they do not allow refinements other than removing nontermination. We use the term "semantic slice" since we are allowing any operation which refines the semantics of the program on the restricted state space. A traditional slice could analogously be called a "syntactic slice" since the $\sqsubseteq$ relation is a purely syntactic one.

Definition 3 A *semantic slice* of $\mathbf{S}$ on X is any program $\mathbf{S}'$ such that:

$$\Delta \vdash \mathbf{S};\ \mathsf{remove}(W \setminus X)\ \leq\ \mathbf{S}';\ \mathsf{remove}(W \setminus X)$$

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, [19] discusses 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. See [18] and [19] for a discussion of the issues.

5.4 Operational Slice

An intermediate option between traditional syntactic slicing and full semantic slicing is to restrict the transformations to preserve operational semantics, using the technique in Section 4.

Definition 4 Program $\mathbf{S}'$ is an *operational slice* of $\mathbf{S}$ on X if there exists a traditional slice $\mathbf{S}''$ of $\mathbf{S}$ on X such that:

$$\Delta \vdash \mathcal{A}(\mathbf{S}'');\ \leq\ \mathcal{A}(\mathbf{S}')$$

A simpler definition, which does not refer to the intermediate program $\mathbf{S}''$ is:

$$\Delta \vdash \mathcal{A}(\mathbf{S}');\ \mathsf{remove}(W \backslash X)\ \leq\ \mathcal{A}(\mathbf{S});\ \mathsf{remove}(W \backslash X)$$

But this is incorrect because the `seq` variable (recording the sequence of states) is one of the variables removed, which means that all of the annotations are redundant code! On the other hand, if we add `seq` to X to stop it from being removed, then the suggested definition is much too restrictive: no statements can be deleted since they all contribute to the value of `seq`.

This is the motivation for introducing the "intermediate" program $\mathbf{S}''$ in our definition of an operational slice.

5.5 Slicing At Any Position

To slice at an arbitrary position in the program we need to preserve the sequence of values taken on by the given variables at that point in the program. To do this, we simply insert an assignment to a new variable slice at the required position which records the current values of the variables. If $X = \{x_1, \ldots, x_n\}$ is the set of variables we are interested in then we insert the statement:

$$\mathsf{slice} := \mathsf{slice} +\!\!+ \langle\langle x_1, \ldots, x_n\rangle\rangle$$

at the point of interest, in order to record the current values of the variables at that point. Then we slice at the end of the program on the single variable slice.

This process can be generalised to slicing at several points in the program, perhaps with a different set of "variables of interest" at each point, simply by inserting the slice assignments at the appropriate places.

One peculiarity of this definition is that if we slice at a point in the program which is within a statement that does not modify any of the variables in the slicing criteria, then we can end up with larger slices than expected. For example, suppose that we slice on x within this **if** statement at the point just before the assignment to z:

$$\textbf{if } y = 0 \textbf{ then } z := 1 \textbf{ fi};$$

According to our definition, the slice has to preserve the test $y = 0$ and therefore preserve any previous modifications to y. In effect, by slicing at a particular position we are insisting that the given *position* should also appear in the sliced program. This is arguably correct in the sense that, if the slice has to preserve the sequence of values taken on by x at a particular point in the program, then a corresponding point (at which x takes on the same sequence of values) must appear in the slice. But if the **if** statement in the above example is deleted, then x takes on a shorter sequence of values! A simple solution to this dilemma is to allow the slicing algorithm to move all the assignments to slice upwards out of any enclosing structures as far as possible, before carrying out the slicing operation itself.

5.6 Dynamic Slicing

Although the term "dynamic program slice" was first introduced by Korel in [Korel 1984], it may be regarded as a non-interactive version of Balzer's notion of flowback analysis [4]. In flowback analysis, one is interested in how information flows through a program to obtain a particular value: the user interactively traverses a graph that represents the data and control dependences between statements in the program. A dynamic slice of a program $\mathbf{P}$ is a reduced executable program $\mathbf{S}$ which replicates the behaviour of $\mathbf{P}$ on a particular initial state. We can define this initial state by means of an assertion. Suppose $V = \{v_1, v_2, \ldots, v_n\}$ is the set of variables in the initial state space for $\mathbf{P}$, and V_1, V_2, $\ldots$, V_n are the initial values of these variables in the state of interest. Then the assertion $\{\mathbf{A}\}$, where $\mathbf{A}$ is $v_1 = V_1 \wedge v_2 = V_2 \wedge \cdots \wedge v_n = V_n$, is a **skip** for this state and **abort** for every other state. We define:

Definition 5 A *Dynamic Slice* of $\mathbf{S}$ with respect to a formula $\mathbf{A}$ of the form

$$v_1 = V_1 \wedge v_2 = V_2 \wedge \cdots \wedge v_n = V_n$$

(where $V = \{v_1, v_2, \ldots, v_n\}$ is the initial state space of $\mathbf{S}$) and the set of variables X (a subset of the final state space W of $\mathbf{S}$) is any program $\mathbf{S}' \sqsubseteq \mathbf{S}$ such that:

$$\begin{aligned}\Delta \vdash \{\mathbf{A}\};\ \mathbf{S};\ \textbf{remove}(W \setminus X)&\\ \leq\ \{\mathbf{A}\};\ \mathbf{S}';\ \textbf{remove}(W \setminus X)&\end{aligned}$$

5.7 Conditioned Slicing

Researchers have generalised dynamic slicing and combined static and dynamic slicing in various ways. For example: some researchers allow a finite set of initial states, or a *partial* initial state which restricts a subset of the initial variables to particular values. In our formalism, all of these generalisations are subsumed under the obvious generalisation of dynamic slicing: why restrict the initial assertion to be of the particular form $\{v_1 = V_1 \wedge v_2 = V_2 \wedge \cdots \wedge v_n = V_n\}$?

If we allow any initial assertion, then the result is called a *conditioned slice*:

Definition 6 A *Conditioned Slice* of $\mathbf{S}$ with respect to any formula $\mathbf{A}$ and set of variables X is any program $\mathbf{S}' \sqsubseteq \mathbf{S}$ such that:

$$\begin{aligned}\Delta \vdash \{\mathbf{A}\};\ \mathbf{S};\ \textbf{remove}(W \setminus X)&\\ \leq\ \{\mathbf{A}\};\ \mathbf{S}';\ \textbf{remove}(W \setminus X)&\end{aligned}$$

One way to construct a conditioned slice is to use the initial condition to simplify the program before applying a traditional slicing algorithm. Danicic et al [9] describe a tool called ConSIT, for slicing a program at a particular point, given that the initial state satisfies a given condition. Conditioned slicing is thus a generalisation of both static slicing (where there are no conditions on the initial state) and dynamic slicing (slicing based on a particular initial state).

The ConSIT tool works on an intraprocedural subset of C using a three phase approach:

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

In ConSIT, the slicing condition can be given in the form of `ASSERT` statements scattered through the program: the authors claim that these `ASSERT` statement are equivalent to a single condition on the initial state, but in general this requires assertions to be formulae of *infinitary* logic. This is because the general case of moving an assertion "backwards" over or out of a loop breaks down into a countably infinite sequence of cases depending on the number of possible iterations of the loop. Fortunately, the assertion statements in WSL are already expressed in infinitary logic, so this is not a problem in our framework.

In our transformation framework, the `ASSERT` statements are simply WSL assertions. The symbolic execution and producing the conditioned program are examples of transformations which can be applied to the WSL program plus assertions. In [14] we provide a number of transformations for propagating assertions and eliminating dead code. Using weakest preconditions, for example, we can move an assertion (with the appropriate modification) backwards past any statement:

$$\Delta \vdash \mathbf{S};\ \{\mathbf{Q}\}\ \approx\ \{\mathrm{WP}(\mathbf{S},\mathbf{Q})\};\ \mathbf{S}$$

For example:

$$x := y+1;\ \{x > 0\}$$

becomes

$$\{y+1>0\};\ x := y+1$$

Similarly, an assertion can be moved out of a loop:

$$\begin{aligned}\Delta \vdash\ & \textsf{while}\ \mathbf{B}\ \textsf{do}\ \{\mathbf{Q}\};\ \mathbf{S}\ \textsf{od}\\ &\approx\ \{\textstyle\bigwedge_{n>0}\big(\bigwedge_{i<n}\mathrm{WP}((\mathbf{S};)^n,\mathbf{B})\\ &\qquad\Rightarrow \mathrm{WP}((\mathbf{S};)^n,\mathbf{Q})\big)\};\\ &\quad \textsf{while}\ \mathbf{B}\ \textsf{do}\ \mathbf{S}\ \textsf{od}\end{aligned}$$

where $(\mathbf{S};)^0$ is **skip** and $(\mathbf{S};)^{n+1}$ is $\mathbf{S};\ (\mathbf{S};)^n$.

Again, a generalisation is suggested: why restrict ourselves to the assertion moving and dead code removal transformations? A conditioned semantic slice can be defined as:

Definition 7 Suppose we have a program **S** and a slicing criterion, defined from **S** by inserting assertions and assignments to the slice variable to form $\mathbf{S}'$. A *conditioned semantic slice* of **S** with respect to this criterion is any program $\mathbf{S}''$ such that:

$$\begin{aligned}\Delta \vdash \mathbf{S}';\ \textsf{remove}(W \setminus \{\textsf{slice}\})\\ \leq\ \mathbf{S}'';\ \textsf{remove}(W \setminus \{\textsf{slice}\})\end{aligned}$$

6 Slicing in FermaT

FermaT is an industrial strength program transformation system, the result of over fifteen years of research and development, which has recently been released under the GNU GPL (General Public Licence). It is available for downloading from:

http://www.dur.ac.uk/∼dcs0mpw/fermat.html

The FermaT syntactic slicer has the following features:

- Handles arbitrary control flow (including WSL code translated from assembler language) via "action systems";
- Interprocedural slicing, which handles the "calling context" problem correctly [10] combined with action systems;
- Efficient algorithms for handling large and complex programs;

FermaT implements a large number of powerful program transformations, these combined with syntactic slicing make it possible to use FermaT for general conditioned semantic slicing.

7 Slicing Example

The following WSL program is a translation of the C program in [6]:

```
$i := 1$;
posprod $:= 1$;
negprod $:= 1$;
possum $:= 0$;
negsum $:= 0$;
while $i \leqslant n$ do
   $a :=$ input[$i$];
   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 $\geqslant$ negsum
                    then possum $:= 0$
                     else negsum $:= 0$ fi;
                 if posprod $\geqslant$ negprod
                    then posprod $:= 1$
                     else negprod $:= 1$ fi fi;
   $i := i + 1$ od;
if possum $\geqslant$ negsum
   then sum := possum
    else sum := negsum fi;
if posprod $\geqslant$ negprod
   then prod := posprod
    else prod := negprod fi
```

Suppose we want to slice this program with respect to the sum variable at the end of the program and with the additional constraint that all the input values are positive. We can either add the assertion $\{\forall i.\, 1 \leqslant i \leqslant n \Rightarrow \mathsf{input}[i] > 0\}$ to the top of the program, or equivalently add the assertion $\{a > 0\}$ just after the assignment to a at the top of the loop. We also append the **remove** statement:

remove(i, posprod, negprod, possum, negsum, n, a, test0)

to the program. This removes all the variables we are not interested in.

The resulting conditioned syntactic slice is:

```
$i := 1$;
possum $:= 0$;
negsum $:= 0$;
while $i \leqslant n$ do
   $a :=$ input[$i$];
   $\{a > 0\}$;
   if $a > 0$
      then possum := possum $+ a$ fi;
   $i := i + 1$ od;
if possum $\geqslant$ negsum
   then sum := possum fi;
remove($i$, posprod, negprod, possum, negsum, $n$, $a$, test0)
```

With semantic slicing we can do much more. For a start, the test $a > 0$ is redundant because of the assertion. Also negsum is zero throughout and possum $\geqslant 0$ throughout (since it is initialised to zero and only modified by having positive numbers added). So a possible semantic slice is:

```
$i := 1$;
possum $:= 0$;
while $i \leqslant n$ do
   $a :=$ input[$i$];
   $\{a > 0\}$;
   possum := possum $+ a$;
   $i := i + 1$ od;
sum := possum;
remove($i$, posprod, negprod, possum, negsum, $n$, $a$, test0)
```

But we can do even more than this. If we first move the assertion out of the loop, then the loop itself can be collapsed to a reduce operation over the input array:

```
$\{\forall i.\, 1 \leqslant i \leqslant n \Rightarrow$ input[$i$] $> 0\}$;
sum := +/input[$1 \,..\, n$];
remove($i$, posprod, negprod, possum, negsum, $n$, $a$, test0)
```

The result is a concise specification of the final value of sum under the given slicing condition.

8 Conclusion

In this paper we have given a brief introduction to the foundations of program transformation theory in WSL and described some applications to program slicing which existing slicing algorithms. Traditional slicing, which is restricted to deleting irrelevant statements has the advantage of a unique solution and may be useful in debugging situations where programmers are already familiar with the layout of the code. But in more general program comprehension, reverse engineering, reengineering and migration tasks, it is much more useful to use transformations to simplify the slices and even present the sliced program at a higher level of abstraction.

A particularly useful application of conditioned semantic slicing is to remove the error handling code during program comprehension or reverse engineering. Often much of the code in a program is there to handle errors: this code can obscure the structure and function of the "main line" code. By adding assertions in appropriate places and slicing on the outputs of interest a much more concise specification of the main function can be generated.

Acknowledgements

The research described in this paper was funded by De Montfort University and Software Migrations Ltd. The idea of initialising v to $\perp$ in a **var** clause where v is used but its initial value isn't suggested by Mark Harman who also pointed out that our method of slicing in the middle of the program arguably creates over-large slices in some cases.

References

[1] R. J. R. Back, *Correctness Preserving Program Refinements*, Mathematical Centre Tracts #131, Mathematisch Centrum, Amsterdam, 1980.

[2] R. J. R. Back, "A Calculus of Refinements for Program Derivations," *Acta Informatica* 25 (1988), 593–624.

[3] R. J. R. Back & J. von Wright, "Refinement Concepts Formalised in Higher-Order Logic," *Formal Aspects of Computing* 2 (1990), 247–272.

[4] R. Balzer, "EXDAMS – EXtendable Debugging And Monitoring System," Proceedings of the AFIPS SJCC, 1969.

[5] Gianfranco Bilardi & Keshav Pingali, "The Static Single Assignment Form and its Computation," Cornell University Technical Report, July, 1999, ⟨http://www.cs.cornell.edu/Info/Projects/Bernoulli/papers/ssa.ps⟩.

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

[7] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

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

[9] 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).

[10] Susan Horwitz, Thomas Reps & David Binkley, "Interprocedural slicing using dependence graphs," *Trans. Programming Lang. and Syst.* 12 (Jan., 1990), 26–60.

[11] C. C. Morgan, *Programming from Specifications*, Prentice-Hall, Englewood Cliffs, NJ, 1994, Second Edition.

[12] C. C. Morgan & K. Robinson, "Specification Statements and Refinements," *IBM J. Res. Develop.* 31 (1987).

[13] Keshav Pingali & Gianfranco Bilardi, "Optimal Control Dependence Computation and the Roman Chariots Problem," *Trans. Programming Lang. and Syst.* (May, 1997), ⟨http://www.cs.cornell.edu/Info/Projects/Bernoulli/papers/toplas97.ps⟩.

[14] M. Ward, "Proving Program Refinements and Transformations," Oxford University, DPhil Thesis, 1989.

[15] M. Ward, "Foundations for a Practical Theory of Program Refinement and Transformation," Durham University, Technical Report, 1994, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/foundation2-t.ps.gz⟩.

[16] M. Ward, "Assembler to C Migration using the FermaT Transformation System," *International Conference on Software Maintenance, 30th Aug–3rd Sept 1999, Oxford, England* (1999).

[17] M. Ward, "Reverse Engineering from Assembler to Formal Specifications via Program Transformations," *7th Working Conference on Reverse Engineering, 23-25th November*, Brisbane, Queensland, Australia (2000), ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/wcre2000.ps.gz⟩.

[18] M. Ward, "Abstracting a Specification from Code," *J. Software Maintenance: Research and Practice* 5 (June, 1993), 101–122, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/prog-spec.ps.gz⟩.

[19] M. Ward, "A Definition of Abstraction," *J. Software Maintenance: Research and Practice* 7 (Nov., 1995), 443–450, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/abstraction-t.ps.gz⟩.

[20] M. Ward & K. H. Bennett, "Formal Methods for Legacy Systems," *J. Software Maintenance: Research and Practice* 7 (May, 1995), 203–219, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/legacy-t.ps.gz⟩.

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