Slicing, I/O and the Implicit State

Yoga Sivagurunathan, Mark Harman and Sebastian Danicic
Project *Project*,
School of Computing,
University of North London,
Eden Grove, London, N7 8DB.
tel: +44 (0)171 607 2789
fax: +44 (0)171 753 7009
e-mail: {eml3sivaguy, m.harman, s.danicic}@uk.ac.unl
www : http://www.unl.ac.uk/~mark/projproj.html

Keywords: Slicing, Embedded Systems, Implicit State, I/O

Abstract

Program slicing consists of deleting statements from a program, creating a reduced program, a slice, that preserves the original program's behaviour for a given set of variables at a chosen point in the program.

However, some aspects of a program's semantics are not captured by a set of variables, rendering slicing inapplicable to their analysis. These aspects of the program's state shall, collectively, be termed the 'implicit state'. For example, the input list supplied to a program is not denoted by a variable, rather it is part of the implicit state. It will be shown that this implicitness causes existing slicing algorithms to produce incorrect slices with respect to input.

In order to solve the problem the program to be sliced will be transformed into an 'explicit' version (in which *all* aspects of its semantics are captured by variables). The approach is also applied to a wider class of problems in which slicing is inhibited by the lack of variables upon which to form a suitable slicing criterion.

Because the approach can be expressed as a source–level transformation, it has the attractive property that the slicing algorithm need not be altered.

1 Introduction

Many programmers spend a considerable amount of time attempting to understand and manipulate computer programs. If the program is sufficiently simple, it can be analysed manually, but such a task is too difficult to perform for larger programs which contain much information which is irrelevant to a particular line of analysis.

Program slicing consists of deleting statements from a program whilst preserving some projection of its semantics, thereby removing such 'irrelevant information'. Slicing has been applied to algorithmic debugging [12], testing[1, 8], integration[10], parallel execution[21], maintenance[6] and measurement[16, 14].

1	`z = 4;`	1	`z = 4;`
2	`y = z + 1;`	2	
3	`x = 5 + z;`	3	`x = 5 + z;`
4		4	
Original Program		Slice w.r.t. (4, {x})	

Figure 1: Weiser's Static Slice

The concept of program slicing was first introduced by Weiser[20, 21]. A wide variety of slicing paradigms have been proposed, each based upon a formulation of the *slicing criterion* (which captures the semantic projection to be preserved during the process of command deletion).

As introduced by Weiser[21], the slicing criterion consists of a line number, n, and a set of variables, S. Consider the program fragment in Figure 1.

The selected variable was `x` and the slicing was performed at (just before the execution of) line `4`. The variable `x` does not depend on `y`, hence the slice does not contain line 2. In this simple example, slicing can be performed by hand. For larger programs, tools such as Unravel [15] can be used to automatically construct slices.

This paper is concerned with the kinds of slices constructed from programs which perform I/O, and, more generally with slices of programs which affect components of the state for which there is no variable to capture the semantic projection of interest. In order to study this prob-

lem the static slicing paradigm will be adopted for simplicity of exposition. However, the results apply equally well to the dynamic [13], quasi–dynamic [19] and conditioned [3, 5] paradigms.

The contribution of this paper can be summarised as follows:

- A minor problem concerning slicing in the presence of input is identified.
- The problem is circumvented using an implicit state removal transformation.
- The transformation is shown to be applicable to a wider class of programs which contain few (or no) variables upon which to form slicing criteria.

The rest of the paper is organised as follows: Section 2 contains some preliminary definitions, which are used in sections 3 and 4 to provide a formal treatment of static slicing in the presence of input statements. Section 5 introduces the implicit state removal transformation, used to rectify a problem identified with slicing in the presence of input, and section 6 shows how this approach can be applied to the more general problem of slicing embedded system programs, which may contain few variables upon which to base a suitable slicing criterion. Section 7 concludes with some directions for future work.

2 Preliminary Definitions

This section introduces some definitions which will be used in subsequent sections.

Definition 1 (REF and DEF Variable Sets) $DEF(n)$ denotes the set of variables defined at node n. $REF(n)$ denotes the set of variables referenced at node n. For example, if n were the assignment statement `x = y + z;` we would have $DEF(n) = \{$`x`$\}$ and $REF(n) = \{$`y,z`$\}$.

Definition 2 (Head and Tail) The head of a sequence, s, shall be denoted $hd(s)$ and the remaining sequence shall be denoted $tl(s)$.

Definition 3 (Function Overriding) This is an operation that takes two functions and creates a new one by overriding all the mappings in the first function with those in the second. We write the overriding of the function, f, by the function g like this: $f \oplus g$.

Definitions 4, 5, 6 and 7 which follow, are those introduced by Weiser [21].

A state trajectory of a program is a trace of its execution, containing 'snapshots' of all its variable values (its state) just before the execution of each statement.

Definition 4 (State Trajectory) A *state trajectory* of length k of a program P is a finite list of ordered pairs

$$< (n_1, s_1), (n_2, s_2), \ldots, (n_k, s_k) >$$

where each n is a node of P and each s is a function mapping the variables in V to their values. Each (n, s) gives the values of variables in V immediately before the execution of n.

Definition 5 (Static Slicing Criterion) A *static slicing criterion* of a program P is a pair $<i, V>$, where i is a statement in P and V is a subset of the variables in P.

A static slicing criterion $C = <i, V>$ determines a projection function Proj_C which throws out of the state trajectory all ordered pairs except those starting with i, and from the remaining pairs throws out all identifiers not in V.

Definition 6 (Projection) Let $T = (t_1, t_2, \ldots, t_n)$ be a state trajectory, n any node in N and s any function from variable names to values. Then

$$\text{Proj}'_{<i,V>}((n, s)) = \begin{cases} \lambda & \text{if } n \neq i \\ (n, s|V) & \text{if } n = i \end{cases}$$

where $s|V$ is s restricted to domain V, and λ is the empty string. Proj′ is now extended to entire trajectories:

$$\text{Proj}_{<i,V>}(T) = \text{Proj}'_{<i,V>}(t_1) \ldots \text{Proj}'_{<i,V>}(t_n).$$

A slice is defined, behaviourally, as any subset of a program which preserves a projection of its behaviour, determined by the slicing criterion.

Definition 7 (Static Slicing) A slice S of a program P on a slicing criterion $C = <i, V>$ is *any* program with the following two properties.

1. S can be obtained from P by deleting zero or more statements from P.
2. Whenever P halts on an input I with state trajectory T, then S also halts on input I with state trajectory T', and $\text{Proj}_C(T) = \text{Proj}_{C'}(T')$, where $C' = <\text{succ}(i), V>$, and $\text{succ}(i)$ is the nearest successor to i in the original program which is also in the slice, or i itself if i is in the slice.

1: x = 4; 2: z = 2*x; 3: q = x; 4: y = z + q; 5:	1: x = 4; 2: 3: q = x; 4: 5:
Original program P	Slice P'

Figure 2: Static Slicing on the Criterion (5, {q})

3 The Formal Definition of a Slice

Suppose the initial state for program P in Figure 2 is σ. The elements of the state trajectory are pairs, (n,σ) where n is the next line to be executed. Therefore at line 1, the pair will be (1,σ). The term 'at line n' means 'when the next line to be executed is at line n'.

In order to define the state trajectory produced by the execution of a program, it will be necessary to formally define the state–to–state mapping[1], $\mathcal{M}_I[\![s]\!]$, denoted by an assignment statement, s. This is defined in the standard way [18], namely:

$$\mathcal{M}_I[\![i{=}e;]\!] = \lambda\sigma.\sigma \oplus \{i \mapsto \mathcal{E}[\![e]\!]\sigma\}$$

Using this semantic description, the state trajectory T, for the example program P can be determined:

$T =$
$<$
(1, σ),
(2, $\sigma\oplus$ {[[x]] $\mapsto$ 4}),
(3, $\sigma\oplus$ {[[x]] $\mapsto$ 4, [[z]] $\mapsto$ 8}),
(4, $\sigma\oplus$ {[[x]] $\mapsto$ 4, [[z]] $\mapsto$ 8, [[q]] $\mapsto$ 4}),
(5, $\sigma\oplus$ {[[x]] $\mapsto$ 4, [[z]] $\mapsto$ 8, [[q]] $\mapsto$ 4, [[y]] $\mapsto$ 12})
$>$

$\text{Proj}_{<n,V>}$ denotes the sequence obtained by removing all pairs (x,y) such that x $\neq$ n, and restricting the state, y, of those which remain, to include only those mappings for variables in V, so $\text{Proj}_{<5,\{q\}>}(T) = <(5, \{[\![q]\!] \mapsto 4\})>$

The state trajectory T' for the slice, P', when the initial state is σ, is:

$T' =$
$<$
(1, σ),
(2, $\sigma\oplus$ {[[x]] $\mapsto$ 4}),
(3, $\sigma\oplus$ {[[x]] $\mapsto$ 4}),
(4, $\sigma\oplus$ {[[x]] $\mapsto$ 4, [[q]] $\mapsto$ 4}),
(5, $\sigma\oplus$ {[[x]] $\mapsto$ 4, [[q]] $\mapsto$ 4})
$>$

Now $\text{Proj}_{<5,\{q\}>}(T') = <(5, \{[\![q]\!] \mapsto 4\})>$, so $\text{Proj}_{<5,\{q\}>}(T) = \text{Proj}_{<5,\{q\}>}(T')$, and therefore P' is a slice of P according to definition 7.

[1]The reason we add the subscript I to $\mathcal{M}$ will become clear in the next two sections.

1: scanf("%d",&x); 2: scanf("%d",&y); 3:	1: scanf("%d",&x); 2: scanf("%d",&y); 3:
Original Program, P	Slice on (3, {y}), P'

Figure 3: Slicing Programs with Input Statements

4 The Input Problem

In the previous section, the state was described as a mapping, $I \rightarrow V$, where I is the set of all variable identifiers and V is the set of all possible values. This form of state is known as an *environment* and V is known as the set of denotable values [18].

In order to represent the semantics of input statements we shall need to augment the environment with a sequence of values, $seq(V)$, to denote the input sequence, thus the state will become $(I \rightarrow V) \times seq(V)$. This augmentation of the state allows us to model the statement scanf("%d",&x); as a state transformation from (σ, i) to ($\sigma \oplus$ {[[x]] $\mapsto$ hd(i)}, tl(i)), enabling us to construct state trajectories for programs which perform input.

Consider, for example the program P in Figure 3. Suppose the slicing criterion is (3, {y}). The state trajectory, T, when the initial environment is σ and the initial input list is i is:

$T =$
$<$
(1, (σ, i)),
(2, ($\sigma\oplus$ {[[x]] $\mapsto$ hd(i)}, tl(i))),
(3, ($\sigma\oplus$ {[[x]] $\mapsto$ hd(i), [[y]] $\mapsto$ hd(tl(i))}, tl(tl(i))))
$>$

Therefore $\text{Proj}_{<3,\{y\}>}(T) = <(3, (\{[\![y]\!] \mapsto \text{hd}(\text{tl}(i))\}, \text{tl}(\text{tl}(i))))>$.

Let P' be a slice of P constructed with respect to the slicing criterion (3, {y}) according to definition 7, and let the state trajectory produced by the execution of P' in the initial state (σ, i) be T'. By definition 7, $\text{Proj}_{<3,\{y\}>}(T)$ must be $\text{Proj}_{<3,\{y\}>}(T')$, so $\text{Proj}_{<3,\{y\}>}(T')$ will be $<(3, (\{[\![y]\!] \mapsto \text{hd}(\text{tl}(i))\}, \text{tl}(\text{tl}(i))))>$.

Clearly therefore, any valid slice, P' of P with respect to (3, {y}) *must* affect the value of the variable y. Since the only statement in P which does this is statement 2, statement 2 must be included in the slice. Furthermore, if the slice P' were to contain only statement 2, then $\text{Proj}_{<3,\{y\}>}(T')$ would be $<(3, (\{[\![y]\!] \mapsto \text{hd}(i)\}, \text{tl}(i)))>$. Therefore, in order to satisfy definition 7, statement 1 must also be included in P'. The only valid slice of P w.r.t. (3,

{y}) is therefore P itself. However, this is not the slice produced by currently published static slicing algorithms [21, 11, 4], (all of which delete line 1).

Existing algorithms fail to produce the correct slice because, according to the standard definition of defined and referenced variables (definition 1),

$$DEF[\![\texttt{scanf("\%d",\&x);}]\!] = \{\texttt{x}\}$$

and

$$REF[\![\texttt{scanf("\%d",\&x);}]\!] = \{\}$$

This means that there will be no *du*–chain [2] between nodes 1 and 2 in the program P in Figure 3. This is an example of a more general problem concerning the 'implicit state' [7, 8, 9].

To see why traditional formulations of defined and referenced variables do not cater for input statements correctly, we need to examine the state in more detail. It will be shown that by making the implicit state explicit the problem can be overcome.

Definition 8 (The Explicit State)
The *explicit* state is $(I \to V)$, where I is the set of variable identifiers and V is the set of denotable values.

Definition 9 (The Implicit State)
The *implicit* state is any part of the state which is not explicit. That is, the implicit state consists of those state components which are not denoted by a variable identifier.

Observe that, whilst the effect of a program has upon the values stored in its variables is explicit, the effect it has upon the input sequence is implicit.

Existing slicing algorithms will include a statement n in a slice iff:

1. the slicing criterion is transitively data dependent on n or,
2. the slicing criterion is transitively control dependent on n.

Data dependence arises because of variable assignments (or, more generally, because of statements which affect the explicit state). Changes to the implicit state do not lead to dependences as there is no variable to carry the dependence. In the most extreme case suppose a non-predicate statement s, affects *only* the implicit state; No slicing criterion can be transitively control or data dependent upon s, and therefore, a slicing algorithm will be free to delete s.

Consider, for example, the program[2] in Figure 4. Suppose the slicing criterion is (3,{y}). The slicing algorithm will be free to delete line 1 because the slicing criterion is neither transitively control nor data dependent upon it. Indeed, line 1 may be deleted in the construction of any slice.

1: getint(); 2: scanf("%d",&y); 3:	1: 2: scanf("%d",&y); 3:
Original Program P	Slice P'

Figure 4: An Incorrect Slice

However, removal of line 1 clearly *does* affect the meaning of line 2. That is, in the original program, line 2 reads the second input into y, whereas, if line 1 is removed, it reads the first. Therefore, removing line 1 will produce a reduced program which does not preserve the effect of the original upon the final value of y. Such a reduced program is not a slice of the original according to Weiser's definition of a slice (definition 7).

5 Denoting the Input Sequence

The solution to the problem lies not in altering the slicing algorithm, rather it requires a change to the value of defined and referenced variable sets (upon which the algorithm depends). This is achieved by a reformulation of the implicit state as an explicit state [7, 8, 9], rather than altering the slicing algorithm, which constructs these slices.

Observe that, because

$$\mathcal{M}_I[\![\texttt{scanf("\%d",\&x);}]\!](\sigma, i) = (\sigma \oplus \{[\![\texttt{x}]\!] \mapsto hd(i)\}, tl(i))$$

it will be inferred that

$$DEF[\![\texttt{scanf("\%d",\&x);}]\!] = \{\texttt{x}\}$$

and

$$REF[\![\texttt{scanf("\%d",\&x);}]\!] = \{\}.$$

because the only variable which alters its value in σ is x and this change references (depends upon) the value of no other mapping in σ. That is, although the input statement affects the implicit state, it does not affect the explicit state. Therefore the defined and referenced variable sets will not capture the linkage between successive input statements; this linkage consists of 'implicit *du*–chains'.

In order to remove the implicit state we need a new variable (and possibly a new denotable value [18] — the list), to denote the implicit state component. In this case, the pseudo-variable Π shall be used to denote the input list.

Let $\mathcal{M}_E$ describe the meaning of a statement in terms of the explicit state.

[2]Where getint() has the sole purpose of consuming an integer from the input.

$$\mathcal{M}_E[\![\texttt{scanf("\%d",\&x);}]\!]\sigma = \sigma \oplus \{[\![\texttt{x}]\!] \mapsto hd(\sigma[\![\Pi]\!]), [\![\Pi]\!] \mapsto tl(\sigma[\![\Pi]\!])\}$$

from which it will be inferred that

$$DEF[\![\texttt{scanf("\%d",\&x);}]\!] = \{\texttt{x}, \Pi\}$$

and

$$REF[\![\texttt{scanf("\%d",\&x);}]\!] = \{\Pi\}.$$

The function Φ, takes an implicit state and transforms into an equivalent explicit state:

$$\Phi : (I \to V) \times seq(V) \longrightarrow (I \to V)$$

$$\Phi(\sigma, i) = \sigma \oplus \{[\![\Pi]\!] \mapsto i\}$$

The connection between $\mathcal{M}_I$ and $\mathcal{M}_E$ is

$$\forall s. \mathcal{M}_I(s) \circ \Phi = \Phi \circ \mathcal{M}_E(s)$$

The relationship between $\mathcal{M}_E$, $\mathcal{M}_I$ and Φ is represented in the commutative diagram below:

$$\begin{array}{ccc} (I \to V) \times seq(V) & \xrightarrow{\mathcal{M}_I(s)} & (I \to V) \times seq(V) \\ \Phi \downarrow & & \downarrow \Phi \\ (I \to V) & \xrightarrow{\mathcal{M}_E(s)} & (I \to V) \end{array}$$

Observe that this reformulation of the implicit state as an explicit state could have been achieved by re–writing the program, introducing assignments to the new pseudo–variable Π. The transformation, $\mathcal{T}$, takes a statement s, and produces a statement s', where s' neither depends upon nor affects implicit state. The transformation $\mathcal{T}$ for our simple `while` loop language is defined in Figure 5.

Observe that $\mathcal{M}_I \circ \mathcal{T} = \mathcal{M}_E$, thus $\mathcal{T}$ is guaranteed to remove the implicit state by source–to–source transformation. This could be established more formally by a simple structural induction on the structure of the language.

6 A Thermostat Program

Often, in embedded real–time systems, there will be a set of primitive commands for controlling input and output using devices such as sensors and actuators. These primitive commands will form part of a control language. Such programs may be hard to slice in any meaningful way, because we shall not be able to identify the interesting properties of the embedded system in slicing criteria — they will all be implicit.

```
1    reset();
2    while (inoperation()) {
3      if (gettemp())
4        switchoff();
5      else switchon();
6      userchoice();
     }
```

Figure 6: A Simple Thermostat Program

Consider, the (highly idealised) thermostat control program in Figure 6. As it stands this program is completely unslicable, as it mentions no variables.

If we model the implicit state using pseudo variables, we shall be able to transform programs such as the thermostat program into longer, but slicable, explicit versions. This corresponds to modelling the unavailable bodies of the primitive functions of the control language. In order to perform this transformation for the thermostat program we will need a specification of the effect of each of the primitives. In this case, the device language primitives control and depend upon a thermometer and a heater. Figure 7 informally specifies the meaning of each primitive of the control language.

Figure 8 describes the implicit state value denoted by each pseudo variable we shall introduce.

Notice that we could, for all such problems, use a *single* variable to capture the entire implicit state [17]. This would require us to model the implicit state as a large data structure, denoted be a single variable. Using a single variable, primitive commands which depend upon or affect the implicit state would be transformed into commands which select and update parts of this data structure. Whilst this approach is theoretically acceptable, it is impractical, as it will dramatically reduce the precision of any slicing algorithm which depends upon it.

Figure 9 describes the transformation function for removing the implicit state. For the `userchoice()` primitive, the enumeration type $\{up, down, manual\}$ is used to model the user's input.

The application of transformation rules from Figure 9 to the program in Figure 6 is depicted Figure 10.

We have adopted a decimal point numbering system to allow us to relate elements of the transformed program to those of the original (via their integral values).

Slicing with respect to (6, {`ideal`}) yields the slice depicted in Figure 11.

Converting this slice back to the original program notation we take the integral part of each statement as the members of the slice, thereby including a statement from the original if any of its transformed counterparts are in the slice of the explicit version.

$$
\begin{array}{rcl}
\mathcal{T}[\![\texttt{scanf}(s,\&i_1,\ldots,\&i_n);]\!] & = & \mathcal{I}[\![i_1]\!]\ldots\mathcal{I}[\![i_n]\!] \\
\mathcal{T}[\![\texttt{while}(e)c]\!] & = & [\![\texttt{while}(e)\{\mathcal{T}[\![c]\!]\}]\!] \\
\mathcal{T}[\![\{c_1\ldots c_n\}]\!] & = & [\![\{\mathcal{T}[\![c_1]\!]\ldots\mathcal{T}[\![c_n]\!]\}]\!] \\
\mathcal{T}[\![\texttt{if}(e)c]\!] & = & [\![\texttt{if}(e)\{\mathcal{T}[\![c]\!]\}]\!] \\
\mathcal{T}[\![\texttt{getint}();]\!] & = & [\![\Pi{=}tl(\Pi);]\!] \\
\mathcal{T}[\![i{=}e;]\!] & = & [\![i{=}e;]\!] \\
 & & \\
\mathcal{I}[\![i]\!] & = & [\![i = hd(\Pi); \Pi = tl(\Pi);]\!]
\end{array}
$$

Figure 5: Removing the Implicit State from a Language with Input Statements

`reset()`	Initialises the ideal temperature setting
`inoperation()`	True iff the user has not switched to manual control
`gettemp()`	True iff the current temperature is ideal (± 2 degrees Fahrenheit)
`switchoff()`	Switches the heater off
`switchon()`	Switches the heater on
`userchoice()`	Allows the user to make one of three decisions: a) to switch to manual control b) to increment the ideal temperature c) to decrement the ideal temperature

Figure 7: Informal Semantics for the Thermostat Language Primitives

Pseudo Variable	Type	Description of Implicit State Modelled
`ideal`	int	The ideal temperature
`IsManual`	boolean	True iff the system is in manual mode
`temp`	int	Current temperature reading on the thermometer
`IsHeaterOff`	boolean	True iff the heater is off
Π	list(choice)	The User's list of inputs

Figure 8: Pseudo–Variables and the Implicit State Components they Denote

$$
\begin{array}{rcl}
\mathcal{T}[\![\texttt{while}(e)c]\!] & = & [\![\texttt{while}(\mathcal{E}[\![e]\!])\{\mathcal{T}[\![c]\!]\}]\!] \\
\mathcal{T}[\![\{c_1\ldots c_n\}]\!] & = & [\![\{\mathcal{T}[\![c_1]\!]\ldots\mathcal{T}[\![c_n]\!]\}]\!] \\
\mathcal{T}[\![\texttt{if}(e)c]\!] & = & [\![\texttt{if}(\mathcal{E}[\![e]\!])\{\mathcal{T}[\![c]\!]\}]\!] \\
\mathcal{T}[\![i{=}e;]\!] & = & [\![i{=}\mathcal{E}[\![e]\!];]\!] \\
\mathcal{T}[\![\texttt{reset}()]\!] & = & [\![\texttt{IsManual} = False; \texttt{ideal} = Default;]\!] \\
\mathcal{T}[\![\texttt{switchoff}()]\!] & = & [\![\texttt{IsHeaterOff} = True]\!] \\
\mathcal{T}[\![\texttt{switchon}()]\!] & = & [\![\texttt{IsHeaterOff} = False]\!] \\
\mathcal{T}[\![\texttt{userchoice}()]\!] & = & [\![\texttt{if}(hd(\Pi) = up)\texttt{ideal} = \texttt{ideal} + 1; \\
 & & \quad \texttt{else if}(hd(\Pi) = down)\texttt{ideal} = \texttt{ideal} - 1; \\
 & & \quad\quad \texttt{else if}(hd(\Pi) = manual)\texttt{IsManual} = True; \\
 & & \quad \Pi = tl(\Pi);]\!] \\
 & & \\
\mathcal{E}[\![\texttt{inoperation}()]\!] & = & [\![\texttt{!IsManual}]\!] \\
\mathcal{E}[\![\texttt{gettemp}()]\!] & = & [\![(\texttt{temp} >= \texttt{ideal} - 2.0)\&\&(\texttt{temp} <= \texttt{ideal} + 2.0))]\!] \\
\mathcal{E}[\![e_1\ b\ e_2]\!] & = & \mathcal{E}[\![e_1]\!]\ b\ \mathcal{E}[\![e_2]\!]
\end{array}
$$

Figure 9: Removing the Implicit State from the Thermostat Control Language

```
1     reset();

2     while (inoperation()) {
3       if (gettemp())
4         switchoff();
5       else switchon();
6       userchoice();




      }
```

Original program

```
1.1   IsManual = False;
1.2   ideal = Default;
2     while (!IsManual) {
3       if ((temp >= ideal - 2.0) && (temp <= ideal + 2.0))
4         IsHeaterOff = True;
5       else IsHeaterOff = False;
6.1     if (hd(Π) = up)
6.2       ideal = ideal + 1;
6.3     else if (hd(Π) = down)
6.4             ideal = ideal - 1;
6.5           else if (hd(Π) = manual)
6.6                   IsManual = True;
6.7     Π = tl(Π);
      }
```

Transformed program

Figure 10: Original and Transformed Thermostat Program

```
1.1   IsManual = False;
1.2   ideal = Default;
2     while (!IsManual) {
6.1     if (hd(Π) = up)
6.2       ideal = ideal + 1;
6.3     else if (hd(Π) = down)
6.4             ideal = ideal - 1;
6.5           else if (hd(Π) = manual)
6.6                   IsManual = True;
6.7     Π = tl(Π);
      }
```

Sliced Transformed Program

```
1     reset();

2     while (inoperation())
6       userchoice();
```

Corresponding Sliced Original Program

Figure 11: Slicing the Thermostat Program Using its Explicit Counterpart

7 Conclusion and Future Work

The implicit–state–removal transformation technique can be applied to any problem where we want to slice a program upon some value which is *implicit*, namely, where there is no variable to denote the state components of interest. The approach is easy to integrate into existing slicing algorithms and tools, as only the defined and referenced variables need change. Furthermore, a suitable change in defined and referenced variables is obtained by pre–transforming the program to be sliced to remove the implicit state. Thus the slicing algorithm can be viewed as an unaffected 'black box process'.

The approach was used to correct a minor deviation of existing slicing approaches concerning the deletion or otherwise of input statements in slice construction. More importantly, it is argued that the approach could be applied to embedded systems, where slicing may be frustrated by a want of variable to slice upon.

More work is required to produce a general set of guidelines for implicit state modelling. It would also be interesting to apply the approach introduced here to other implicit state components. For example, file systems and dynamic memory allocation and deallocation. Such analysis might prove fruitful in producing more precise slices of programs which affect and depend upon the heap store, and may allow us to address problems associated with programs which perform I/O through file access. This work could also allow previously implicit computation to be analysed using slicing. For example, the potential of a program to leak dynamic memory could be analysed by making the implicit top of heap explicit.

References

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

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, techniques and tools*. Addison Wesley, 1986.

[3] G. Canfora, A. Cimitile, Andrea De Lucia, and G. A. Di Lucca. Software salvaging based on conditions. In *International Conference on Software Maintenance (ICSM'96)*, pages 424–433, Victoria, Canada, September 1994. IEEE.

[4] Sebastian Danicic, Mark Harman, and Yogasundary Sivagurunathan. A parallel algorithm for static program slicing. *Information Processing Letters*, 56(6):307–313, December 1995.

[5] Andrea. De Lucia, Anna Rita Fasolino, and Malcolm Munro. Understanding function behaviours through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension*, Berlin, Germany, March 1996.

[6] Keith B. Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[7] Mark Harman. *Functional Models of Procedural Programs*. PhD thesis, University of North London, 1992.

[8] Mark Harman and Sebastian Danicic. Using program slicing to simplify testing. *Journal of Software Testing, Verification and Reliability*, 5:143–162, September 1995.

[9] Mark Harman and Sebastian Danicic. Slicing programs in the presence of errors. *Formal Aspects of Computing*, 1996. To appear.

[10] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating non–interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[11] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 25–46, Atlanta, Georgia, June 1988. Proceedings in *SIGPLAN Notices*, 23(7), pp.35–46, 1988.

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

[13] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[14] Arun Lakhotia. Rule–based approach to computing module cohesion. In *Proceedings of the* 15^{th} *Conference on Software Engineering (ICSE-15)*, pages 34–44, 1993.

[15] James R. Lyle, Dolores R. Wallace, James R. Graham, Keith B. Gallagher, Joseph P. Poole, and David W. Binkley. Unravel project.

[16] Linda M. Ott and J. J. Thuss. The relationship between slices and module cohesion. In *Proceedings of the* 11^{th} *ACM conference on Software Engineering*, pages 198–204, May 1989.

[17] D. A. Schmidt. *Denotational semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.

[18] Joseph E. Stoy. *Denotational semantics: The Scott–Strachey approach to programming language theory*. MIT Press, 1985. Third edition.

[19] G. A. Venkatesh. The semantic approach to program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–28, Toronto, Canada, June 1991. Proceedings in *SIGPLAN Notices*, 26(6), pp.107–119, 1991.

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

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