Analysis of Dynamic Memory Access using Amorphous Slicing

Mark Harman,
Department of Mathematical and Computing Sciences,
Goldsmiths College, University of London, Lewisham Way, New Cross, London SE14 6NW, UK.
Tel: +44 (0)171 919 7850. Fax: +44 (0)171 919 7853.
M.Harman@gold.ac.uk

Yoga Sivagurunathan and Sebastian Danicic
School of Informatics, University of North London, Eden Grove, London, N7 8DB.
Tel: +44 (0)171 607 2789 Fax: +44 (0)171 753 7009
yogas@motivforce.com, s.danicic@unl.ac.uk

Abstract

Problems associated with understanding, verifying and re-engineering the way in which a system allocates and releases dynamic memory present significant challenges to the software maintainer. Because the questions underlying these problems are undecidable, no system can provide a completely fail safe certification. For example, in checking for memory leaks, a system can only warn of potential problems, but cannot guarantee that no leaks remain.

We present an approach to modelling the dynamic memory access properties of a program using amorphous program slicing to create a Dynamic Memory Model *(*DMM*). The slices are constructed from a transformed version of the original program in which heap access has been made explicit using a pseudo variable to denote the top of the heap. The* DMM *is a simplified version of the original program which is concerned solely with the dynamic memory access behaviour of the original. We illustrate the use of* DMMs *in problems of comprehension, verification and re-engineering.*

We introduce a proof-of-concept DMM *construction algorithm, showing how slicing simplification power can be dramatically improved using domain-specific transformation rules.*

1. Introduction

Most systems allocate and deallocate memory as they execute. This dynamic memory allocation is often hard to understand and problems associated with dynamic memory are notoriously difficult to model, predict and verify. For example, the cause of the London Ambulance Service System's second crash at 2am on November 4, 1992 was blamed on the accidental insertion of a memory leak in the vehicle mobalisation routine [10] (section 4039).

In this paper we introduce an approach modelling dynamic memory access properties using a mixture of implicit state removal [22] and amorphous slicing [15]. The focus of this work is an analysis of the quantity of dynamic memory allocated, rather than analysis of other dynamic memory access problems such as dangling pointers and inappropriate de-referencing.

To motivate our approach consider the program fragment in the leftmost section of Figure 1, where `N` is a symbolic compile-time constant. The central section of the figure contains the version of the program with an additional variable `hp`, which stores the current value of the heap top. Finally the rightmost section contains the amorphous slice constructed with respect to the final value of the pseudo variable `hp`. We call this form of amorphous slice a Dynamic Memory Model(DMM). The DMM captures the computation in the original program concerned with dynamic memory access and removes other code which is irrelevant. For example, in this case, the DMM clearly indicates the amount of heap store used by the original program.

This paper contains two related contributions to the problem of understanding, verifying and re-engineering the dynamic memory access properties of a program:

1. Extracting and understanding dynamic memory properties is reformulated as a program simplification problem.

   This enables us to divide the problem into an automated phase (program simplification) and a non-automated phase (reasoning about the simplified program). This separation will be necessary in any complete approach, as the underlying questions concerning dynamic memory consumption are undecidable.

   The philosophy of 'programs as approximate answers to undecidable analysis questions' is an underlying theme of this research.

2. We show that the specific approach to simplification we ad-

Corrected Original	Implicit state removed	Dynamic Memory Model

`rec = NULL;`
`i = 0;`
`while(i<N)`
`  { n = malloc(sizeof(struct L));`
`    n->next = rec;`
`    n->data = i;`
`    rec = n;`
`    i=i+1; }`
	`hp = 0;`
	`rec = NULL;`
	`i = 0;`
	`while(i<N)`
	`  { hp = hp + sizeof(struct L);`
	`    n = malloc(sizeof(struct L));`
	`    n->next = rec;`
	`    n->data = i;`
	`    rec = n;`
	`    i=i+1; }`
		`hp = N*sizeof(struct L) ;`

Figure 1. A Simple Motivating Example

vocate makes a set of powerful simplifying transformations applicable.

This is important because defining suitable transformation strategies remains an open problem. In this case we have a simple example of 'domain specific program transformation', where the domain knowledge is used to guide the transformation strategy. When used in a more generic transformation scenario, these domain specific transformations would be unlikely to yield such dramatic simplification.

The rest of this paper is organised as follows:- Section 2 defines an algorithm that makes the heap pointer explicit and section 3 briefly reviews amorphous program slicing, arguing that it is better suited to this problem than conventional slicing alone. Section 4 introduces some transformation rules for amorphous slicing of DMMs, showing how domain knowledge can be exploited to greatly improve simplification power. Section 5 presents a simple algorithm for computing amorphous DMM slices, which is illustrated with a worked example in section 6. Section 7 shows how DMMs can be applied to comprehension, analysis and re-engineering problems. Section 8 concludes with directions for future work.

2. Modelling the Implicit State

Our first step is to make the dynamic memory properties of the program 'sliceable'. This is not possible with the original code, because there is no variable which denotes the amount of heap allocated. More precisely, there is no explicit state component which captures this – there is, of course, an *implicit* top of heap pointer which does so.

We use the term 'implicit state' for components of the state which do not form a part of the store. We view a store as a mapping from names to values, thus it has the properties of an environment in a functional language [23]. We use the term 'explicit state' to refer to this store, because its components have an explicit denotation (the variables) in the program and can thus form the subject of a slicing criterion.

In [22] we introduced a simple technique for denoting implicit state components with pseudo variables, making these components explicit. This transformation of an implicit-state program into an explicit-state program is useful because it allows us to form a slicing criterion which captures the (previously implicit) state. The approach, its correctness and semantic foundations are described in more detail in [22].

The implicit state removal algorithm for a simple intraprocedural language (involving heap allocation using `malloc`) is presented as a context-free, syntax directed transformation in Figure 2. The effect of this simple linear–time source-to-source transformation is to insert assignments into the program to a new variable, `hp`, which ensure that `hp` stores the current amount of heap store allocated. Thus all programs start with an initial assignment of `hp = 0;`, and where ever there is an allocation of memory in the original program, the transformed version will contain a re-assignment to `hp`.

We have made some simplifying assumptions about the kind of program to which this transformation will be applied. For example, we assume that calls to `malloc` can only occur as the right hand sides of assignments and that the actual parameter to `malloc` does not, itself, contain a call to `malloc`. These merely serve to simplify the exposition; a more complex implicit-state removal transformation can be defined to handle the more general case where `malloc` calls can occur in arbitrary expressions.

Also, we assume that expressions are side effect free.

3. Amorphous Slicing

Having introduced the pseudo variable `hp` to capture the implicit computation on the top of the heap, we are left with a program which is longer than that with which we started. However, the explicit version of the program is amenable to slicing.

A conventional program slice [25, 16, 14] is an abstraction of a program which contains the computation associated with some set of variables at some point of interest. The pair (V, n), containing the set of variables of interest, V and the point of in-

$$
\begin{array}{rcl}
p & \in P & \text{(programs)} \\
c, c_i & \in C & \text{(commands)} \\
e, e_i & \in E & \text{(expressions)} \\
b & \in B & \text{(boolean operators)} \\
v, v_i & \in I & \text{(variable names)} \\
s & \in S & \text{(strings)} \\
\mathcal{T} & : & P \to C \\
\mathcal{C} & : & P \cup C \to C \\
\mathcal{E} & : & E \to C \\
 & & \\
\mathcal{T}[\![p]\!] & = & [\![\texttt{hp} = 0;\;\; \mathcal{C}[\![p]\!]]\!] \\
 & & \\
\mathcal{C}[\![\texttt{while}(e)\ c]\!] & = & [\![\texttt{while}(e)\{\mathcal{C}[\![c]\!]\}]\!] \\
\mathcal{C}[\![\{c_1 \ldots c_n\}]\!] & = & [\![\{\mathcal{C}[\![c_1]\!] \ldots \mathcal{C}[\![c_n]\!]\}]\!] \\
\mathcal{C}[\![\texttt{if}(e)\ c]\!] & = & [\![\texttt{if}(e)\{\mathcal{C}[\![c]\!]\}]\!] \\
\mathcal{C}[\![\texttt{if}(e)\ c_1\texttt{else}\ c_2]\!] & = & [\![\texttt{if}(e)\{\mathcal{C}[\![c_1]\!]\}\ \texttt{else}\ \{\mathcal{C}[\![c_2]\!]\}]\!] \\
\mathcal{C}[\![i{=}e;]\!] & = & [\![\mathcal{E}[\![e]\!];\, i{=}e;]\!] \\
\mathcal{C}[\![\texttt{for}(e_1;e_2;e_3)c]\!] & = & [\![\texttt{for}(e_1;e_2;e_3)\mathcal{C}[\![c]\!]]\!] \\
\mathcal{C}[\![\texttt{scanf}(s,v_1,\ldots,v_n);]\!] & = & [\![\texttt{scanf}(s,v_1,\ldots,v_n);]\!] \\
\mathcal{C}[\![v- >e_1{=}e_2;]\!] & = & [\![\mathcal{E}[\![e_2]\!];v- >e_1{=}e_2;]\!] \\
 & & \\
\mathcal{E}[\![\texttt{malloc}(e)]\!] & = & [\![\texttt{hp} = \texttt{hp}{+}e;]\!] \\
\mathcal{E}[\![e_1\ b\ e_2]\!] & = & \mathit{append}(\mathcal{E}[\![e_1]\!], \mathcal{E}[\![e_2]\!])
\end{array}
$$

Figure 2. Making the Heap Top Explicit

terest, n is known as the slicing criterion. The process of slicing consists of identifying all statements and predicates which can have no effect upon the value of any variable in V when the next statement to be executed is at point n. A slice is thus both a syntactic and a semantic projection of the program from which it is constructed. Detailed surveys of program slicing can be found in papers by Tip [24] and Binkley and Gallagher [5]. There is also an excellent 'slicing home page' maintained by Jens Krinke [19].

Amorphous slicing [15] is a variation of conventional slicing in which the slice must preserve the effect of the original program with respect to the slicing criterion, whilst it need not be a syntactic subset of the original. Amorphous slicing therefore retains the semantic aspect of conventional slicing, but removes the syntactic restriction to command deletion.

Definition 1 (Conventional Slice)
A conventional slice, s of a program p is constructed with respect to a slicing criterion, (V, n).
(**semantic property**)
For all variables v in V, and for all initial states, the value of v at point n in s must be identical to the value of v at point n in p.
(**syntactic property**)
s is a conventional slice of p iff it respects the semantic property and it is obtained from p by statement deletion[1].

Definition 2 (Amorphous Slice)
An amorphous slice, s of a program p is constructed with respect to a slicing criterion, (V, n).
(**semantic property**)
For all variables v in V, and for all initial states, the value of v at point n in s must be identical to the value of v at point n in p.
(**syntactic property**)
An amorphous slice is constructed with respect to a simplicity measure[2], $\sqsubseteq$, which orders programs according to some notion of simplicity.

s is an amorphous slice of p iff it respects the semantic property and $s \sqsubseteq p$.

In this paper, we shall only require amorphous slices to be constructed for the single variable hp at the end of the program, so the amorphous slicing criterion will always be $(\{\texttt{hp}\}, L)$, where L is the end node of the program's CFG.

The relationship between amorphous and conventional slicing is discussed in more detail in [15]. Here, we simply note that amorphous slices are never thicker and often considerably thinner than corresponding conventional slices. The extra simplicity is achieved because the amorphous slice need not syntactically resemble the original program. This makes the approach inappropriate for some applications of slicing (for example cohesion measurement [21, 3], and debugging [20]).

However, slicing has also been applied to comprehension [11, 18] and reuse [7, 9] and for these applications amorphous slicing is more suitable, as slice simplicity is more important than syntactic preservation. Indeed, in cases where we have previously applied an implicit state removal transformation, we will have already deviated from the syntax of the original source code, and so there is nothing to be lost by applying amorphous slicing in place of conventional slicing.

To construct a DMM of a program p, we use amorphous slicing at the end of $\mathcal{T}[\![p]\!]$ with respect to the variable hp. This produces a program which is as simple as possible but which has the same effect as $\mathcal{T}[\![p]\!]$ on the variable hp. Thus, the DMM captures in hp the effect of p on the top of the heap.

Definition 3 (DMM)
Let p be a program or program fragment. The DMM of p is the amorphous slice constructed for the program $\mathcal{T}[\![p]\!]$ with respect to the slicing criterion $(\{\texttt{hp}\}, L)$, where L is the end node of the CFG of $\mathcal{T}[\![p]\!]$.

4. Domain Specific Transformation

Transformation rules suitable for creating DMMs from our simple language are listed in Figure 3. The rules are written in the form of a quasi logical calculus, so they are not fixed to

[1] or, in many slicing implementations, by node *identification* in the PDG [17].

[2] In this paper we shall use a node count, that is

$$p \sqsubseteq q \Leftrightarrow \#(\mathrm{dom}(\mathrm{CFG}(p))) \leq \#(\mathrm{dom}(\mathrm{CFG}(p)))$$

where $\#$ is set cardinality and $\mathrm{dom}(\mathrm{CFG}(x))$ is the domain of the CFG of the program x.

the transformation strategy we use in this paper (presented in section 5).

A rule of the form: $\frac{A}{B \Rightarrow C}$ can be interpreted as "If A holds then the fragment B can be transformed to the fragment C".

The terms $\mathbf{REF}(e)$ and $\mathbf{DEF}(e)$ denote the referenced and defined [1] variables of e (we employ a slight abuse of notation by assuming that these two functions may be applied to either expressions or to commands). The term $\mathbf{SUB}(e_1, i, e_2)$ returns the expression that results from substituting all occurrences of the variable i in the expression e_1, with the expression e_2.

Three of these transformations ('Collapse Incremental For Loop', 'Collapse Decremental For Loop' and 'Collapse If-Else') are worthy of special note. They resemble strength reducing operations on expressions used in peep-hole compiler optimisation [1] and in partial evaluation [6]. The collapsing transformations reduce the strength of statements, by identifying situations in which non-primitive statements (`for` or `if`) can be replaced by a simple assignment.

Because of their simplifying power, these collapsing rules would be very useful in program analysis were they to be widely applicable. However, in most 'normal' programs they will be far from widely applicable and so a *general* transformation strategy would be unlikely to make progress using them.

However, the programs we shall be constructing amorphous slices from are far from normal. We claim that it is *not* unlikely that a programmer will write (perfectly sensible) code that will make these two rules applicable when applied to the problem of constructing DMMs.

Conditionals

It not unlikely that the programmer will write code which contains a conditional in which the same amount of memory is allocated in both the `then` and `else` branches (even though this memory may be put to entirely different uses). In such cases the Collapse If-Else axiom will be applicable, once the implicit state removal transformation, $\mathcal{T}$, has been applied.

Loops

It is not unlikely that a programmer will write a loop to create some recursive data structure (such as a list or tree). At each execution of the loop a constant amount of memory will be consumed. If the loop executes for a bounded number of iterations (as is the case with many `for` loops) and this bound can be inferred as a compile-time expression, then transformations like the `for` loop collapsing rules will become applicable.

5. A Simple Algorithm for Intraprocedural Amorphous DMM Slicing

We have implemented a prototype tool, based upon a simple algorithm which produces DMMs from single procedure programs. In its current form the tool merely serves to demonstrate 'proof of concept', as it handles neither the interprocedural case, nor aliasing.

Nonetheless, this simple algorithm is capable of producing the amorphous slices presented in this paper, and we believe that it demonstrates the feasibility of and justification for the concept of a Dynamic Memory Model.

The algorithm is presented in Figure 4, a larger example is presented in Figure 5 and two ancillary transformation tactics are described in Figures 6 and 7. The essential idea is to use the push and pull rules to expand expressions and to coalesce assignments to the same variable, in a manner reminiscent of symbolic execution strategies. This has the effect of reducing the interdependencies between statements at the expense of enlarging their expressions. The reduction in inter-dependencies increases the deletion opportunities available to conventional slicing.

Having simplified the program by expanding expressions and slicing away the unwanted statements, the collapsing rules and axioms may become applicable. If they are, then they will introduce additional assignment statements (in place of conditionals and loops) and these new assignments can be pushed using the 'symbolic execution' phase. These 'pushes' in turn may make more slicing and collapsing possible. The algorithm iterates until no further collapsing rules apply.

Termination of the algorithm is guaranteed because the collapsing rules guarantee to reduce program size and if none apply then the algorithm halts. The algorithm essentially has quadratic complexity because it involves a pass of (potentially) the whole program for each assignment statement. This complexity makes the algorithm of value only for intraprocedural amorphous slicing, as to perform interprocedural amorphous slicing on large programs would be prohibitively expensive. Fortunately, work is in progress [4] on an implementation of amorphous slicing using transformations on program dependence graphs, which will produce a more realistic interprocedural algorithm.

6. Worked Example

Consider the "exam marks" program in the leftmost section of Figure 5. The result of removing the implicit state (Step 1) is given in the central section of the figure and the result of constructing an initial conventional slice (Step 2) is given in the rightmost section.

The conventional slice is the starting point for the iterative steps of the algorithm. It is not required for correctness, but as this example demonstrates, conventionally slicing the program does tend to remove a lot of code which serves to reduce the effort required by the transformation/slicing phase which follows it.

Executing Step 3.1 has the effect of collecting together the two assignments to `hp` inside the `for` loop (using the Unfold Assign rule). This transforms[3] the DMM to:

```
hp = 0;
scanf("%d",&num) ;
```

[3] Our prototype tool does reduce constant expressions, though it currently performs no other expression level simplification.

Rule 1 (Push Assignment)

$$\frac{i_1 \neq i_2, i_2 \notin \mathbf{REF}(e_1), e_3 = \mathbf{SUB}(e_2, i_1, e_1)}{[\![i_1 = e_1;\ \ i_2 = e_2;]\!] \Rightarrow [\![i_2 = e_3;\ \ i_1 = e_1;]\!]}$$

Rule 2 (Unfold Assignment)

$$\frac{e_3 = \mathbf{SUB}(e_2, i, e_1)}{[\![i = e_1;\ \ i = e_2;]\!] \Rightarrow [\![i = e_3;]\!]}$$

Rule 3 (Push If)

$$\frac{e_2' = \mathbf{SUB}(e_2, i, e_1), [\![i{=}e_1;\ c]\!] \Rightarrow [\![c'\ \ i{=}e_1;]\!]}{[\![i{=}e_1;\ \texttt{if}\ \ (e_2)\ \ c]\!] \Rightarrow [\![\texttt{if}\ \ (e_2')\ \ c'\ \ i{=}e_1;]\!]}$$

Rule 4 (Push If-Else)

$$\frac{e_2' = \mathbf{SUB}(e_2, i, e_1),\ \ [\![i{=}e_1;\ \ c_1]\!] \Rightarrow [\![c_1'\ \ i{=}e_1;]\!],\ \ [\![i{=}e_1;\ \ c_2]\!] \Rightarrow [\![c_2'\ \ i{=}e_1;]\!]}{[\![i{=}e_1;\ \ \texttt{if}\ \ (e_2)\ \ c_1\ \ \texttt{else}\ \ c_2]\!] \Rightarrow [\![\texttt{if}\ \ (e_2')\ \ c_1'\ \ \texttt{else}\ \ c_2'\ \ i{=}e_1;\ \ ]\!]}$$

Rule 5 (Collapse For Loop[a])

$$\frac{(\{i\} \cup \mathbf{DEF}(e_1) \cup \mathbf{DEF}(e_2) \cup \mathbf{DEF}(e_3)) \cap \mathbf{REF}(e) = \emptyset}{[\![\texttt{for}\ \ (e_1;\ \ e_2;\ \ e_3)\ \ i{=}e;]\!] \Rightarrow [\![i{=}e;]\!]}$$

Rule 6 (Collapse Incremental For Loop)

$$\frac{\{i_1, i\} \cap \mathbf{REF}(e_3) = \emptyset}{[\![\texttt{for}\ \ (i{=}e_1;\ \ i{<}e_2;\ \ i{=}i{+}1)\ \ i_1{=}i_1{+}e_3;]\!] \Rightarrow [\![i_1{=}i_1{+}(e_2{-}e_1){*}e_3;]\!]}$$

Rule 7 (Collapse Decremental For Loop)

$$\frac{\{i_1, i\} \cap \mathbf{REF}(e_3) = \emptyset}{[\![\texttt{for}\ \ (i{=}e_1;\ \ i{<}e_2;\ \ i{=}i{+}1)\ \ i_1{=}i_1{-}e_3;]\!] \Rightarrow [\![i_1{=}i_1{-}(e_2{-}e_1){*}e_3;]\!]}$$

Rule 8 (Pull Read)

$$\frac{c \text{ contains no } \texttt{scanf} \text{ statements}, i \notin (\mathbf{REF}(c) \cup \mathbf{DEF}(c))}{[\![c\ \ \texttt{scanf}(\text{"}\%\texttt{d}\text{"}, \&i);]\!] \Rightarrow [\![\texttt{scanf}(\text{"}\%\texttt{d}\text{"}, \&i);\ \ c]\!]}$$

Rule 9 (`while` to `for`)

$$\frac{i \notin (\mathbf{DEF}(c) \cup \mathbf{REF}(e))}{[\![i{=}e_1;\ \ \texttt{while}(i{<}e_2)\{c\ \ i{=}i{+}e;\}]\!] \Rightarrow [\![\texttt{for}(i{=}e_1;\ \ i{<}e_2;\ \ i{=}i{+}e)\ \ c]\!]}$$

Axiom 1 (Collapse If-Else) $[\![\texttt{if}(e)\ \ c\ \ \texttt{else}\ \ c]\!] \Rightarrow [\![c]\!]$

[a] Loop rules assume that the loop is executed at least once

Figure 3. Transformation Rules for Amorphous DMM Slices

Step 1	Introduce a variable `hp` using the transformation, $\mathcal{T}$, in Figure 2
Step 2	Construct a conventional slice for the criterion $(\{\texttt{hp}\}, L)$
Step 3.1	Use SE to symbolically execute (see Figure 6)
Step 3.2	Construct the conventional slice for the criterion $(\{\texttt{hp}\}, L)$
Step 3.3	Apply the collapsing rules and `while` to `for` where possible
Step 3.4	If any collapsing rule was applicable then repeat from Step 3.1

Figure 4. The Top Level algorithm

```
pass = NULL;
fail = NULL;
passnum=0;

printf("Number of students?");
scanf("%d",&num) ;

for (i=0;i<num;i=i+1) {
  e = (exam) malloc(32);
  printf("\n%d name:  ",i+1);
  scanf("%s",&name);
  printf("\n%d mark:  ",i+1);
  scanf("%d",&marks);
  strcpy(e->name,name);
  e->marks = marks;
  r = (list) malloc(10) ;
  r->datum = e;

  if (marks >= 40) {
    r->next = pass;
    pass = r;
    passnum = passnum+1;
    }
  else {
    r->next = fail;
    fail = r;
    }
  }

printf("\n\n");
for (i=0;i<passnum;i=i+1) {
  e = pass->datum;
  printf("\nName:  %s",e->name);
  printf("\nMark:  %d",e->marks);
  pass = pass->next; }
```

Original

```
hp = 0;
pass = NULL;
fail = NULL;
passnum=0;
printf("Number of students?");
scanf("%d",&num);

for (i=0;i<num;i=i+1) {
  hp = hp + 32;
  e = (exam) malloc(32);
  printf("\n%d name:  ",i+1);
  scanf("%s",&name);
  printf("\n%d mark:  ",i+1);
  scanf("%d",&marks);
  strcpy(e->name,name);
  e->marks = marks;
  hp = hp + 10;
  r = (list) malloc(10) ;
  r->datum = e;
  if (marks >= 40) {
    r->next = pass;
    pass = r;
    passnum = passnum+1;
    }
  else {
    r->next = fail;
    fail = r;
    }
  }
printf("\n\n");
for (i=0;i<passnum;i=i+1) {
  e = pass->datum;
  printf("\nName:  %s",e->name);
  printf("\nMark:  %d",e->marks);
  pass = pass->next; }
```

Implicit State Removed

```
hp = 0;




scanf("%d",&num);

for (i=0;i<num;i=i+1) {
  hp = hp + 32;








  hp = hp + 10;















  }
```

Conventional Slice

Figure 5. Exam Marks Program

```
SE : listof(C) → listof(C)

SE(sl) is defined as follows
set WorkList := sl, DoneList := ∅, TopList := ∅
while WorkList not empty do
 if head(WorkList) is a for, if or while then
  Apply SE to the body of head(WorkList)
 else
  if head(WorkList) is an assignment then
   set (c, cl, stuck) :=
      PUSH(head(WorkList),tail(WorkList))
   if not(stuck) then
    set WorkList := cl
    Append c to DoneList
   else Append c to TopList fi
  else Append c to TopList fi
 fi
 set WorkList := tail(WorkList) od
return Append(TopList, DoneList)
```

Figure 6. Symbolic Execution Algorithm

```
PUSH : C × listof(C) → C × listof(C) × bool

PUSH(c, cl) is defined as follows
set WorkList := cl, DoneList := ∅
while WorkList not empty do
 set ⟦i = e⟧ := c
 if head(WorkList) is an assignment then
  set ⟦i′ = e′⟧ := head(WorkList)
  if i = i′ then
   return (∅, Append(DoneList, WorkList), true)
  else
   if Push Assignment rule applies then
    set ⟦c1; c⟧ := result of Push Assignment rule
    Append c1 onto the end of DoneList
   else
    return (c, Append(DoneList, WorkList), true) fi
  fi
 else if head(WorkList) is a conditional then
   if Push If or Push If-Else rules apply then
    set ⟦p; c⟧ := result of Push rule
    Append p onto the end of DoneList
   else
    return (c, Append(DoneList, WorkList), true) fi
  else
   return (c, Append(DoneList, WorkList), true) fi
  set WorkList := tail(WorkList) fi
od
return (c, DoneList, false)
```

Figure 7. Push Algorithm

```
for(i=0;i<num;i=i+1)
  hp = hp+42;
```

Step 3.2 (conventional slicing) has no effect upon this transformed program, but Step 3.3 (the Collapsing stage) reduces it to:

```
hp = 0;
scanf("%d",&num) ;
hp = hp+(num-0)*42;
```

Since some collapsing took place, Step 3.4 returns us to Step 3.1, which produces:

```
scanf("%d",&num) ;
hp = 0+(num-0)*42;
```

Conventional slicing (Step 3.2) and the collapsing rules (Step 3.3) have no effect, so the algorithm terminates with step 3.4.

It is interesting to note that although this DMM is attractively small, its very simplicity could be regarded as a disadvantage; for example, we have lost the `printf` statement which accompanied the `scanf` statement and which to some extent played the role of a comment in the way it described the value read in by the `scanf`. This unfortunate 'loss of context' is a property of all slicing-based approaches to simplification. It is a property which we could ameliorate using origin tracking [13]. This is an approach used by the PIM system [12], which constructs slices by term rewriting. Each syntactic term maintains a link to the context (its origin) from which it came prior to the application of the rewrite rule which produced it.

Using the algorithm in Figure 4, we have thus constructed from the original program a small, simple model of its behaviour with regard to the heap store.

At this point it is only fair to make clear that the dramatic simplification that we have been able to achieve is possible because the programs we have considered contained loops for which the number of iterations could be determined as a compile-time expression. However, while this may not always be the case, we shall clearly be able to exploit this property in any fragments where it does apply. Also, as might be expected, we have found that many of the programs we have examined contain a large amount of code which does not contribute to the heap access characteristics of the overall program. This guarantees that slicing will be an attractive technique for focusing upon those parts of the program which do involve the heap.

7. The Use of DMMs in Analysis

In this section we consider a few applications to which DMMs may be put. The list is not exhaustive, as a DMM is simply a semantic abstraction of a program's heap computation and therefore can be used in any task which requires analysis of this behaviour.

7.1. Re-Engineering

In some cases it may be possible to use the DMM to identify re-engineering opportunities. It is likely that some sections of code will allocate an amount of dynamic memory which is bounded and for which the upper bound can be deduced at compile time. From the original program, it will not be immediately obvious which sections of code have this property and for those which do, it will not be obvious what the upper bound is. Similarly, we might like to deduce a lower bound for memory allocation.

Using the DMM, the bounds for any allocation of dynamic memory will be easier to deduce, because all computation not involving memory allocation will be stripped away and that which remains will 'homogenise' all memory allocations as simple integer operations on `hp`. As all slicing algorithms are conservative, we cannot guarantee that all such 'irrelevant' computation will be removed. However, our algorithm is safe; if code is not present in the DMM then such code is definitely irrelevant.

7.2. Space Complexity Formulae

In addition to extracting lower and upper bounds on memory allocation, it will also be useful to construct a formula which expresses the amount of store used by a subsystem in terms of some determining parameters. This is a hard problem for a maintainer presented with an entire system, but the DMM presents this information in isolation and in the familiar syntax of the original programming language.

For example, suppose a large system contains a fragment of code like that depicted in Figure 8. This fragment creates a table, allocating row elements with `AllocateRow` and column elements with `AllocateCol`, each of which takes a pointer and creates enough space for the structure it references. The functions `RowCalc` and `ColCalc` establish the values of row and column elements in terms of the current row number and in terms of the current row and column number respectively.

Suppose the amount of store consumed by a call to `AllocateRow` is r cells and that consumed by `AllocateCol` is c cells. Observe that rows are allocated in the outer loop (which executes n times), whereas column elements are allocated in the inner loop (which executes nm times). We could reasonably expect that the fragment will consume $rn + cnm$ store cells in total.

The maintainer could attempt to establish this for themselves by manual inspection. However, this would be a time-consuming and costly activity. On the other hand, using our DMM construction algorithm, we obtain the DMM shown in the rightmost section of Figure 8, revealing at a glance the space complexity of the original code fragment.

We could simplify this DMM further by applying simple algebraic identities to simplify expressions. We have not yet written such expression-level simplification rules into our prototype tool. This form of expression simplification is included in comparable systems such as PIM [12] and the maintainers' assistant [2, 8], and we plan to augment our tool in a similar fashion, to improve the quality of the DMMs it produces.

7.3. Identifying Memory Leaks

Consider the program fragment in the leftmost section of Figure 9. In the original fragment `getmem()` is a function which allocates `const` bytes of store, while `releasemem(s)` returns `const` bytes of store to the free list starting at address `s`. The part of the fragment marked with three vertical dots indicates an arbitrary section of code which neither allocates nor releases dynamic memory.

The central section of the figure contains the result of removing the implicit state and the final section contains the DMM. Once again the ability to simplify expressions would improve the algorithm; we would be able to transform the assignment `hp = (const+(N-1)*const)-(N-1)*const;` to produce the final DMM:

$$\mathtt{hp} = \mathtt{const};$$

... demonstrating the presence of a memory leak of `const` bytes even more starkly.

Of course this simple program only allocated memory. We have not covered dynamic memory deallocation in the implicit state removal transformation, $\mathcal{T}$, presented in Figure 2. To do so is not hard, but it does require the retention of a symbol table, which complicates the exposition somewhat. More work is required to define implicit state removal for more elaborate memory management systems and to explore the transformation sets which are most appropriate to the models which arise.

8. Conclusion and Future Work

We argue that slicing (and in particular amorphous slicing) is applicable to the problem of analysing dynamic memory for possible problem computations, such as those which lead to leaks. We show that a Dynamic Memory Model (DMM) can be constructed using amorphous slicing on a program which has been pre-transformed into a version in which the top of the heap is explicit.

The domain-specific nature of the approach means that powerful simplification transformations become applicable, making the size of DMMs surprisingly small. It appears that similar applications of amorphous slicing will also need to use domain-specific transformations to ameliorate the problem of identifying suitable transformation strategies.

Unfortunately slices in general and static slices in particular have a tendency to be rather large; often important computations touch a large section of the original program code. Amorphous slicing attempts to address this problem by focusing upon the semantics of the subcomponents to be 'sliced away' rather than the syntax in which they are expressed. Clearly, if the semantic

Original Program	Dynamic Memory Model

`hp = 0 ;`
`for(i=0;i<n;i=i+1)`
`{ AllocateRow(NewRow) ;`
`Row = AppendRow(Row,RowCalc(NewRow,i)) ;`
`for(j=0;j<m;j=j+1)`
`{ AllocateCol(NewCol) ;`
`Col = AppendCol(Col,ColCalc(NewCol,i,j))`
`}`
`}`
	`hp = (n-0)*(r+(m-0)*c) ;`

Figure 8. Using DMMs to Capture Space Complexity of Underlying Algorithms

Original	After Implicit State Removal	Dynamic Memory Model

`s = getmem();`
`s->data = Sentinel ;`
`/* this sentinel marks */`
`/* the end of the list */`
`s->next = NULL ;`
`for(i=1;i<N;i=i+1)`
`{ t = getmem();`
`t->data=i;`
`t->next=s;`
`s=t; }`
⋮
`for(i=1;i<N;i=i+1)`
`{ t = s->next ;`
`releasemem(s) ;`
`s=t; }`
	`hp = 0;`
	`hp = hp+const;`
	`s = getmem();`
	`s->data = Sentinel;`
	`s->next = NULL;`
	`for(i=1;i<N;i=i+1)`
	`{ hp = hp+const;`
	`t = getmem();`
	`t->data=i;`
	`t->next=s;`
	`s=t; }`
	⋮
	`for(i=1;i<N;i=i+1)`
	`{ t = s->next;`
	`hp = hp-const;`
	`releasemem(s);`
	`s=t; }`
		`hp = (const+(N-1)*const)-(N-1)*const;`

Figure 9. Using DMMs to Identify Memory Leaks

projection concerned is inexpressible with simple syntax, then amorphous slicing also will produce a large slice.

One solution to the problem of finding good (i.e. small) slices therefore consists of finding applications where the sub component of interest has a low 'semantic mass'. (By 'semantic mass' we mean the smallest amount of syntax required to express some semantic feature.) We believe that typical programs *will* enjoy a low semantic mass with respect to the amount of dynamic memory allocated, making amorphous slicing highly applicable. This remains a conjecture, until further empirical studies can be conducted on production software.

Also, we have only considered a memory model in which memory is dynamically allocated in a C-like programming style. Our prototype tool for DMM construction would also benefit from some additional knowledge of algebraic equalities on side-effect free expressions and from a wider class of 'collapsing' transformations. These practical extensions and the empirical validation of the 'low semantic mass hypothesis' remain problems for future work.

Acknowledgements

We would like to thank the anonymous referees for their helpful comments, which lead to a re-write of several sections and which, we hope, we have used successfully to improve the presentation of this work.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, techniques and tools*. Addison Wesley, 1986.

[2] K. Bennett and M. Ward. Formal methods to aid the evolution of software. *Journal of Software Engineering and Knowledge Engineering*, 5(1):25–47, 1995.

[3] J. M. Bieman and L. M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, Aug. 1994.

[4] D. Binkley. Private communication, 1998.

[5] D. W. Binkley and K. B. Gallagher. Program slicing. In M. Zelkowitz, editor, *Advances of Computing, Volume 43*. Academic Press, 1996.

[6] D. Bjørner, A. P. Ershov, and N. D. Jones. *Partial evaluation and mixed computation*. North–Holland, 1987.

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

[8] W. C. Chu, P. Luker, and H. Yang. Code understanding through program transformation for reusable component identification. In 5^{th} *IEEE Internation Workshop on Program Comprehesion (IWPC'97)*, Dearborn, Michigan, USA, May 1997.

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

[10] Communications Directorate, South West Thames Regional Health Authority. Report of the inquiry into the London Ambulance Service, ISBN: 0 905133 70 6, Feb. 1993. Also available on The Risks Digest, Volume 14, Issue 4, 7 April, 1993 (http://catless.ncl.ac.uk/Risks/14.48.html).

[11] A. De Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviours through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension*, Berlin, Germany, Mar. 1996.

[12] J. Field, G. Ramalingam, and F. Tip. Parametric program slicing. In 22^{nd} *ACM Symposium on Principles of Programming Languages*, pages 379–392, San Francisco, CA, 1995.

[13] J. Field and F. Tip. Dynamic dependence in term rewriting systems and its application to program slicing. In M. Harman and K. Gallagher, editors, *Journal of Information and Software Technology Special Issue on Program Slicing*, volume 40. Elsevier, 1998. to appear.

[14] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, Aug. 1991.

[15] M. Harman and S. Danicic. Amorphous program slicing. In 5^{th} *IEEE Internation Workshop on Program Comprehesion (IWPC'97)*, pages 70–79, Dearborn, Michigan, USA, May 1997.

[16] S. Horwitz, T. Reps, and D. 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.

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

[18] D. Jackson and E. J. Rollins. Chopping: A generalisation of slicing. Technical Report CMU-CS-94-169, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1994.

[19] J. Krinke. Program slicing page, URL: `http://www.cs.tu-bs.de/~krinke/`.

[20] J. R. Lyle and M. Weiser. Automatic program bug location by program slicing. In 2^{nd} *International Conference on Computers and Applications*, pages 877–882, Peking, 1987.

[21] L. 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.

[22] Y. Sivagurunathan, M. Harman, and S. Danicic. Slicing, I/O and the implicit state. In M. Kamkar, editor, 3^{rd} *International Workshop on Automated Debugging (AADEBUG'97)*, pages 59–65, Linköping, Sweden, May 1997.

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

[24] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.

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