VADA: A Transformation-based System for Variable Dependence Analysis

Mark Harman[1] **Chris Fox**[2] **Rob Hierons**[1] **Lin Hu**[1] **Sebastian Danicic**[3]
Joachim Wegener[4]

[1]Brunel University
Uxbridge, Middlesex
UB8 3PH, UK.

[2]Essex University
Colchester
CO4 3SQ, UK.

[3]Goldsmiths College
New Cross, London
SE14 6NW, UK.

[4]DaimlerChrysler
Alt-Moabit 96a
D-10559 Berlin, Germany

Keywords: Program Slicing, Program Transformation, Variable Dependence Analysis

Abstract

Variable dependence is an analysis problem in which the aim is to determine the set of input variables that can affect the values stored in a chosen set of intermediate program variables.

This paper shows the relationship between the variable dependence analysis problem and slicing and describes VADA*, a system that implements variable dependence analysis.*

In order to cover the full range of C constructs and features, a transformation to a core language is employed. Thus, the full analysis is required only for the core language, which is relatively simple. This reduces the overall effort required for dependency analysis. The transformations used need preserve only the variable dependence relation, and therefore need not be meaning preserving in the traditional sense. The paper describes how this relaxed meaning further simplifies the transformation phase of the approach. Finally, the results of an empirical study into the performance of the system are presented.

1 Introduction

VADA is a system for variable dependence analysis. The system is designed to augment the DAIMLER-CHRYSLER Evolutionary Testing System [42]. The variable dependence problem is that of determining the set of variables V' at point n' in a program that can affect the values of a selected set of variable V at point n. The pair (V, n) is referred to as the dependence criterion, or simply, the 'criterion'.

For example consider the code fragment in Figure 1. This figure depicts a simple program with a single

```
while(a>1)
  { c=c-1;
    b=b+1;
    x=x+y+b;
    y=y+z;
    p=p+1;
    z=z+1; }
```

Figure 1. Variable Dependence Example

`while` loop. In this example, the variable `x` at the end of the program depends upon the initial values of the variable set $\{\texttt{z}, \texttt{y}, \texttt{b}, \texttt{a}, \texttt{x}\}$. The example indicates the way in which variable dependence can be loop carried (the dependence of `x` upon `y`) and involves control dependence (the dependence of `x` upon `a`) as well as data dependence (all other dependences in this example).

Variable dependence information can be used to augment the effectiveness of evolutionary testing. Evolutionary testing [26, 31, 32, 33, 37] uses search-based techniques to find good quality test data by searching the set of possible inputs. Test data quality is defined by a test adequacy criterion, which underpins the fitness function that drives the search implemented by the evolutionary algorithm.

The computational effort required to find a suitable test input (that causes execution to follow a required branch) is closely related to the size of the space to be searched. For each branch there is a controlling predicate. The size of the search space is defined by the number of input variables considered to be relevant to the computation of this predicate. Of course, not all input variables contribute to the computation of every predicate. There will be some predicates that depend upon relatively few input variables. Clearly, in such cases, it is wasteful to search the entire input space, when only a small portion can possibly contain the required test input. This is the problem addressed by VADA. The approach adopted is similar to the chaining approach of Ferguson and Korel [17].

This paper describes the design and implementation of the VADA system, and in particular, its combination of transformation and analysis and its use of memoisation. The principal contributions of the paper are as follows:

1. An illustration of the relationship between slicing and variable dependence.

2. The description of a core language for C and an algorithm for transforming C into this core.

3. A formal description of a slicing and variable dependence algorithm for the core language.

4. An illustration of the benefits of this two phase transform and analyse approach.

5. An empirical study into the performance improvement produced by memoisation.

The core language has two useful properties. Compared to a treatment of full C, it is relatively easy to

1. transform C into the core language

2. compute slices and variable dependence for the core language

The rest of this paper is organised as follows. Section 2 shows the relationship between slicing and variable dependence. Sections 3 and 4 describe the overall architecture of the VADA system and the core language used by the slicing engine at its heart. Sections 5 and 6 describe the variable dependence analysis and transformation phases of the system in more detail and Section 7 presents some initial empirical results concerning the performance of the analysis phase.

2 The Relationship Between Variable Dependence and Slicing

The problem of variable dependence is closely related to program slicing [14, 20, 44] and chopping [25] because it involves the traversal of transitive data and control dependence relations. This section shows that the result of variable dependence analysis can be used to compute slices and that slicing is also useful in the computation of variable dependence. In our implementation, variable dependence for programs expressed in the whole language is computed in terms of slices constructed for the transformed program in the core language.

2.1 Slicing for Variable Dependence Analysis

Program slicing consists of identifying the parts of a program that can potentially affect the values of a chosen set of variables [6, 14, 20, 36, 44]. There are various forms of slicing including the original static formulation [43], and subsequent dynamic [1, 27], quasi-static [38], conditioned and constrained [7, 10, 18] and pre/post conditioned [21]. Slices can be construed to be syntax preserving or amorphous [5, 19, 40].

The problem of computing static variable dependence can be partly solved by a solution to the corresponding static slicing question. This result can be used in order to determine control flow. That is, in order to decide whether `p` is in the control dependence relation for the variable `x` in `if (p==0) S`, it is necessary to determine whether any of the statements of `S` contribute to the computation of the value of `x`. This can be answered by slicing `S` with respect to `x`. If the slice is empty, then the value of `x` does not depend upon the value of `p`, otherwise it does.

2.2 Variable Dependence for Slicing

Danicic [9] shows that slicing and variable dependence are related: a slice can be computed from a variable dependence relation by a suitable instrumentation of the source program to be sliced. The instrumentation involves inclusion of an additional referenced variable in each expression. This is a unique variable, v, which occurs nowhere else in the program. Therefore, if v is included in the variable dependence for some variable v' then the expression tagged by v is a member of the slice on v'. The identification of expressions (both arithmetic and boolean) required in a slice is sufficient to determine the slice itself. That is, the nodes of the Control Flow Graph typically denote expressions (either the boolean expressions of conditional statements or the arithmetic expressions of assignments).

For example, consider the program in Figure 2(a). Static slices are typically computed for a schema in which the information available determines the reference variables of an expression, but not the function mapping denoted by the expression [9]. The slice for the schema in Figure 2(b) will therefore be valid of any instantiation of functions for f_3, f_4 and f_5 and predicates b_1 and b_2.

For instance, the final value of `x` depends upon the pseudo variables $\{N_1, N_2, N_3, N_4, N_5\}$, indicating that the slice on `x` is the entire program. The variable `i` depends upon the pseudo variables N_1 and N_5 only indicating that the slice on `i` is the `while` loop and assignment to `i`.

`while (i<0) {`
`  if (c==3) {`
`    c=y;`
`    x=x+5; }`
`  i=i+1; }`
	`while (`b_1`(i)) {`
	`  if (`b_2`(c)) {`
	`    c=`f_3`(y);`
	`    x=`f_4`(x); }`
	`  i=`f_5`(i); }`
		`while (`b_1`(`N_1`,i)) {`
		`  if (`b_2`(`N_2`,c)) {`
		`    c=`f_3`(`N_3`,y);`
		`    x=`f_4`(x,`N_4`); }`
		`  i=`f_5`(`N_5`,i); }`

(a) Program	(b) Schema	(c) With Expression Labelling Pseudo Variables

Figure 2. The Correspondence Between Slicing and Variable Dependence

3 The VADA System Architecture

The overall VADA system is split into two subcomponents: a transformer and an analyser. The transformer converts C source code into its representation in the core language, which becomes the subject of the analysis phase. This follows the approach first suggested by Landin [29] and that has been widely used since.

The transformation phase preserves the variable dependence relation (though not necessarily the full meaning) of the original. The pre-transformation serves to simplify the role of the analyser. The overall system accepts C, except that recursion and backward `goto` statements are not currently supported. The system also assumes side–effect free expressions but work is currently in progress to augment the system with side effect removal [22] as a part of the transformation phase.

VADA uses SWI-prolog[1] to implement a syntax-directed variable dependence algorithm, derived from the parallel slicing algorithm of Danicic and Harman [11]. Prolog was chosen because it allows speedy prototyping of new features and because it has elegant facilities for processing the XML parse tree (produced by a proprietary third party parser licensed to DaimlerChrysler).

4 The Core Language

VADA's the core language consists of a simple subset of intraprocedural C. It is fully analysed to produce both slices and variable dependence information. The subset does not restrict expression notation, but does restrict procedural abstraction and only supports a limited set of statement constructs. The syntax of the core language is defined in Figure 3. The syntactic class I is the class of C program identifiers.

A program in the core language consists of a set of function definitions, the bodies of which are the statement sequences of statements drawn from the syntactic class C. The core allows for function definition but not function call. Function calls are unfolded in the transformation phase. Theoretically, this can lead to exponential growth in program size, but in practice this problem has yet to be experienced by the authors; cases of

[1]SWI-Prolog is available under the GNU Public Licence from `http://www.swi.psy.uva.nl/projects/SWI-Prolog/`.

$$
\begin{array}{rcl}
E & ::= & E_1\,\mathtt{BOp}\,E_2 \\
 & | & !(E) \\
 & | & I \\
 & | & \mathrm{N} \\
 & | & I[E] \\
 & | & E_1 ? E_2 : E_3 \\
\mathtt{BOp} & ::= & + \mid - \mid * \mid \% \mid\ == \mid != \mid \mathtt{\&\&} \mid || \ldots \\
 & & \\
C & ::= & I = E; \\
 & | & \mathbf{if}(E)\ C \\
 & | & \mathbf{if}(E)\ C_1\ \mathbf{else}\ C_2 \\
 & | & \mathbf{while}(E)\ C \\
 & | & \{C_1\ \ldots\ C_n\} \\
 & | & \mathbf{goto} L; \\
 & | & L :
\end{array}
$$

Figure 3. The Syntax of the Core Language

exponential growth appear to be result of purely 'pathological examples', which, though theoretically possible, seldom appears to arise in practice. The syntax of declarations is omitted for brevity; declarations only play a role where structured types are concerned, since all scalar types are considered to be equivalent at the level of abstraction dealt with by VADA.

The core language was chosen to facilitate easy translation from full C, without disturbing the control flow relation of the full language. That is, the `while` loop and `if` statement are retained as the paradigms of selection and repetition, so that all conditional and iteration constructs can be converted into these core constructs, while preserving the control dependence relation. A smaller core language would have been possible [40], but the objective was to retain a simple approach to control dependence, in which a predicate controls the statements mentioned in its body. This greatly reduces the effort required to compute control dependence information needed by the slicing and dependence analysis algorithm for the core language.

5 The Slicer/Variable Dependence Analyser for the Core Language

The prolog rules for slicing the core language yield variable dependence information as a by-product. The rules are attractively simple to define; the essence of the slicing engine is presented in Figure 4. The auxiliary predicates (for which detail is not provided) are described in English in Figure 5. Space does not allow the full details of these rules to be given. However, they are relatively straightforward and the reader can easily reconstruct all but the `empty_statement` predicate, the definition of which determines the precision of the slicing algorithm in the presence of unreachable code. The current implementation makes some attempt to recognize unreachable code (which should be removed from all slices regardless of the slicing criterion), but does not detect all cases. Of course, in general, determination of whether or not a piece of code is unreachable is undecidable [2], but it remains open as to whether unreachable code can be determined at this level of abstraction [9, 30, 45].

The parsing details are left abstract in the exposition; rather than cluttering the details of the algorithm with constructors and selectors for parsing, a form of syntactic representation using Quine's quasi quotes $[\![\ldots]\!]$ has been adopted. These quotes (or 'syntactic brackets') are typically used to distinguish the language under analysis from the language used to perform the analysis [35].

Although Prolog allows the expression of pseudo non-deterministic algorithms via backtracking, our simple slicing algorithm is fully deterministic, and hence has no need to exploit this Prolog feature[2].

The predicate `slice` has six arguments. Three are input arguments and three are output arguments. These arguments retain this distinct role throughout the computation of slices and variable dependence.

The predicate call

```
slice(MapIn,MapOut,C,CritIn,C',CritOut)
```

takes a statement `C` and an input criterion (a set of variables of interest) `CritIn`, and returns a slice `C'` and an output criterion (a set of variables upon which `CritIn` depends), `CritOut`. The predicate `slice` also takes a mapping from labels to slicing criteria, `MapIn` and produces an updated mapping, `MapOut`.

The slice `C'` is the end slice [28] of the statement `C` with respect to the set of variables `CritIn`. `CritOut` is the set of variables, the initial values of which (prior to execution of `C`) determine the values of the variables in `CritIn` after the execution of C. The mapping, `MapIn`, records the slicing criterion that was computed for the labels as they are encountered. This mapping is updated as new labels are discovered. In the computation of fixed points (for the computation of the variables set `CritOut` for `while` loops) the mapping is also updated for previously encountered labels that are re-encountered as the iteration to the fixedpoint proceeds.

Each of the rules for computing slices are relatively straightforward. The assignment and conditional rules implement, in Prolog, the traditional dataflow equations used for slicing [11, 44]. Each rule is processed according to Prolog's first fit pattern-matching rules, so that the rule that keeps a statement in the slice is considered first, and only if this fails is the statement removed, by the clause that follows.

The rule for slicing a `while` loop deserves a little further explanation. The rule first slices the body of the loop. If this is empty, then the slice should remove the entire `while` loop, hence the 'not empty' test in the second line of the first clause for `while` loops. However, if the loop is to be retained, it is necessary to compute the smallest set of variables that contains the input criterion (`CritIn`) and that remains invariant after the loop body is completed. The predicate `find_fixpoint` has the role of computing this criterion, storing the result in `FixPointCrit`. However, the value of `FixPointCrit` is not the ultimate resulting criterion, as the loop may not be executed (so the initial `CritIn` should be retained), and should always contain the predicate referenced variables (so `PredRefs` is retained). There is a more elegant method of handling loops in terms of unfolded, nested `if` statements, but this is less efficient.

The rules for labels (labels are treated as statements by the parser) and `goto` statements, merely record and restore, respectively, the criterion associated with the label mentioned in the statement.

6 The Transformation Phase

The transformation phase transforms C programs into the core language, the syntax of which was defined in Figure 3. The overall transformation algorithm is as follows:

Step 1	Insert Dependence Criteria Assignments
Step 2	Globalise local scope
Step 3	Build function and procedure symbol table
Step 4	Unfold procedure calls
Step 5	Globalise local scope
Step 6	Statement-level transformation rules

The remainder of this section describes each step in more detail.

Step 1 inserts assignment statements to pseudo variables to capture the criteria specified by the user in an XML variable dependence criterion. These assignments allow dependence at arbitrary points in the program to be computed in terms of corresponding dependences at the end of the program. For example, to insert a criterion

[2] As the algorithm is deterministic, the implementation can make extensive use of the cut operator (`!`) to eliminate irrelevant "choice points," and hence reduce memory consumption. However, these details are not relevant to the values of slices and variable dependence computed and so they are omitted from the exposition.

```
slice(MapIn,MapIn,⟦I = E;⟧,CritIn,⟦I = E;⟧,CritOut) :-
      intersection(CritIn,[ I ],Intersection),
      not(Intersection = []),
      collect_refs(⟦E⟧, VarNamesR),
      subtract(CritIn,VarNamesL,Difference),
      union(Difference,VarNamesR,CritOut).
slice(MapIn,MapIn,⟦I = E;⟧,CritIn,⟦⟧,CritIn).

slice(MapIn,MapOut,⟦if(E) C⟧,CritIn,⟦if(E) C′⟧,CritOut) :-
      slice(MapIn,MapOut,⟦C⟧,CritIn,⟦C′⟧,CritThen),
      not(empty_statement(⟦C′⟧)),
      collect_refs(⟦E⟧ ,PredRefs),
      big_union([CritIn,CritThen,PredRefs],CritOut).
slice(MapIn,MapOut,⟦if(E) C1 else C2⟧,CritIn, ⟦if(E) C1′ else C2′⟧,CritOut) :-
      slice(MapIn,ThenMapOut,⟦C1⟧,CritIn,⟦C1′⟧,CritThen),
      slice(MapIn,ElseMapOut,⟦C2⟧,CritIn,⟦C2′⟧,CritElse),
      (not(empty_statement(⟦C1′⟧));not(empty_statement(⟦C2′⟧))),
      collect_refs(⟦E⟧,PredRefs),
      union(ThenMapOut,ElseMapOut,MapOut),
      big_union([CritThen,CritElse,PredRefs],CritOut).
slice(MapIn,MapIn,⟦if(E) C⟧,CritIn,⟦⟧,CritIn).

slice(MapIn,MapOut,⟦while(E) C⟧,CritIn,⟦while(E) C′⟧,CritOut) :-
      slice(MapIn,_MapOut,⟦C⟧,CritIn,⟦C″⟧,FirstCritOut),
      not(empty_statement(⟦C″⟧)),
      collect_refs(⟦E⟧,PredRefs),
      find_fixpoint(MapIn,_MapOut',PredRefs,[],FirstCritOut,⟦C⟧,_CritIn',FixPointCrit),
      union(FixPointCrit,CritIn,CritOfSlicedBody),
      slice(MapIn,MapOut,⟦C⟧,CritOfSlicedBody,⟦C′⟧,CritOutFromSlicedBody),
      big_union([CritOutFromSlicedBody,PredRefs,CritIn],CritOut).

slice(MapIn,MapIn,⟦while(E) C⟧,CritIn,⟦⟧,CritIn) :-

slice(MapIn,MapOut,⟦{C1 ... Cn−1 Cn}⟧,CritIn,{C1′ ... C′n−1 Cn′},CritOut) :-
      slice(MapIn,MapInternal,⟦Cn⟧,CritIn,⟦Cn′⟧,CritInternal),
      slice(MapInternal,MapOut⟦{C1 ... Cn−1}⟧,CritInternal,⟦{C1′ ... C′n−1}⟧,CritOut).

slice(MapIn,MapIn,⟦goto L;⟧,_CritIn,⟦goto L;⟧,CritOut) :-
      lookupMap(MapIn,⟦L⟧,CritOut).
slice(MapIn,MapOut, ⟦L :⟧,CritIn,⟦L :⟧,CritIn) :-
      updateMap(MapIn,⟦L⟧,CritIn,MapOut).


find_fixpoint(MapIn,MapIn,_PredRefs,PrevButOneCrit,PrevCrit,_Body,_CritIn,PrevCrit) :-
      subset(PrevCrit,PrevButOneCrit).

find_fixpoint(MapIn,MapIn,PredRefs,_PrevButOneCrit,PrevCrit,Body,CritIn,FinalCrit) :-
      big_union([PrevCrit,PredRefs,CritIn],NewCritIn),
      slice(MapIn,MapOut',Body,NewCritIn,_Body2,CurrentCrit),
      find_fixpoint(MapOut',MapOut,PredRefs,PrevCrit,CurrentCrit,Body,CritIn,FinalCrit).
```

Figure 4. Core Language Slicing Algorithm in Prolog Pseudo Code

Auxiliary Prolog Predicate	Purpose
`big_union`	Forms the distributed union of a set of sets
`collect_refs`	Determines the referenced variables of an expression
`empty_statement`	Determines whether a statement contains only dead code and empty structures
`lookupMap`	Find the criteria to which a label is mapped
`updateMap`	Update a map to record a new mapping for a Label to a Criterion

Figure 5. Details of Auxiliary Predicates used by the Slicer/Variable Dependence Analyser

$\{v_1, \ldots, v_n\}$, an assignment $\pi = \pi + v_1 + \ldots + v_n$ is inserted at the point of interest, where π is a pseudo variable not otherwise used in the program. The fact that the expression assigned to this pseudo variable involves additions reflects and arbitrary choice of operator required in order to ensure that the variables in $\{v_1, \ldots, v_n\}$ are referenced. The pseudo variable itself, is also included as a referenced variable in order to correctly compute dependence on $\{v_1, \ldots, v_n\}$, where the point of interest lies inside a loop.

Step 2 creates new, previously unused variables and uses these to replace, consistently, the occurrences of local variables, thereby removing local scope and ensuring that all variables are, effectively, global variables. This simplifies the variable analysis phase, particularly in the presence of `break` and `return` statements. Subsequent transformation steps replace both `break` and `return` statements with forward jumps, but they retain the property that they allow criteria to pass from an outer to an inner scope. Without globalisation, the scope rules would have to be taken into account every time such a statement is encountered and, potentially, separate treatments would be required for each variable, depending upon its scope binding.

Step 3 is a pre-requisite for Step 4. It is not really a transformation step because it does not affect the program, but it is required in order to record the mapping from the name of a procedure to its body.

Step 4 uses the procedure symbol table that was created in step 3 to unfold each call to a procedure call in the bodies of procedures. Clearly this is not possible in the presence of recursion. However, VADA was designed for use with the DAIMLERCHRYSLER Evolutionary Testing System, which is typically applied to embedded systems, which rarely, if ever, contain recursion.

Unfolding procedures may seem like a wasteful and rather crude transformation to perform. However it has a number of advantages that make it worthwhile in practice. Specifically, the subsequent analysis phase is not only considerably easier to express, it is also faster. This extra speed is obtained at the expense of space, but not precision. The slicer (which produces variable dependence as a by product) need only be intraprocedural; the unfolding of procedure calls takes account of the calling context by introducing assignments to the formal parameters from the actual parameters of the call. This means that during the analysis there is no additional overhead to take account of calling context. A similar association of formals to actuals is achieved in standard approaches to slicing [24], without the overhead of copying procedure bodies. However, these approaches involve the construction of a dependence graph for the entire language under consideration, which is a considerably larger undertaking than the simple analysis required by VADA for its core language. It is not obvious how these dependence graph based approaches could be easily adapted to provide variable dependence information, rather than slices.

Step 5 repeats the globalisation process. This is required because Step 4 creates additional local scope when formal parameters are transformed into assignments to new local variables. This second globalisation process has to take place after the unfolding of procedures in order to correctly account for formal parameters. However, the globalisation of procedure bodies also has to take place *before* unfolding, in order to ensure that after unfolding, each mention of the same local variable is mapped to the same global variable, regardless of the call instance. Therefore, globalisation has to be performed twice: once before unfolding and once after.

Step 6 performs statement level transformations. These simplify language under consideration. For instance, the C language offers many ways of assigning a value to a variable, through assignment expressions, using assignment operators like `+=` and `&=` and through the use of pre- and post- increment and decrement operators. Sequencing of assignments within expressions is also possible using the comma operator. When such expressions are encountered as statements, they are each transformed into 'regular' assignment statements of the form `I = E;`. The `do` and `for` looping constructs are converted to `while` constructs. This means that only a single looping construction need be considered in the analysis phase. This is a saving in development effort, as correct account of looping constructs requires a fixpoint computation, as explained in Section 5 and memoisation, as explained in Section 7. It also has a positive benefit on testing an verification, since the transformation steps can be readily understood and verified in isolation, leaving only a single looping construct to be tested.

The 'jump' statements, `break`, `continue` and `return` are converted to forward `goto` statements and labels are introduced in order to capture the targets of these `goto` statements. This allows all these three forward jumps[3] to be treated identically in the analysis phase. This unification of statement construct offers similar benefits for understanding, development and verification as the unification of looping constructs.

However, all of these transformations are unremarkable and well-studied in the literature and folklore of transformation. What was more surprising was to discover that transformations need not be meaning preserving, at least, not in the traditional sense of meaning—that is typically considered to be some form of functional equivalence [3, 4, 13, 34, 39, 41]. Although non-meaning preserving transformations have been considered for program modification in corrective and adaptive maintenance [15, 16], the use of non-meaning preserv-

[3]A `return` becomes a forward jump after unfolding.

ing transformation here appears to be novel. That is, for the purpose of producing the core language from the C language, it is only necessary to preserve sufficient aspects of the behaviour of the program to ensure that the variable dependence relation remains invariant throughout the transformation. The authors believe that this may turn out to be a significant and useful departure from traditional program transformation. This is reminiscent of the recent trend in abstract interpretation research away from applications to analysis and towards manipulation [8].

6.1 Ease of Handling Semantically Complex Constructs: the `switch` Statement

The transformation function $\mathcal{R}^{\mathrm{L}}(C)$ produces a statement or statement sequence C' from a statement of statement sequence C. C' is identical to C except that all occurrences of the `break` statement have been replaced by the statement `goto L;`. The transformation of a `switch` statement converts the `switch` into a nested conditional, using $\mathcal{R}^L(C)$, to replace `break` statements with forward jumps that exit the nested structure. More formally, the `switch` statement **switch** (E) $\{$ **case** c_1 : C_1 ... **case** c_n : C_n **default** : d $\}$ is transformed to the nested conditional

$$
\begin{array}{l}
\textbf{if}\,(E) \\
\quad \{\ \textbf{if}\,(1) \\
\qquad \{\ \textbf{if}\,(1) \\
\qquad \vdots \\
\qquad\quad \{\ \textbf{if}\,(1) \\
\qquad\qquad \{\ \textbf{if}\,(1)\,\mathcal{R}^L(C_1)\ \ \} \\
\qquad\qquad \mathcal{R}^L(C_2)\ \ \} \\
\qquad\qquad \vdots \\
\qquad \mathcal{R}^L(C_{n-1})\ \ \} \\
\quad \mathcal{R}^L(C_n)\ \ \} \\
\textbf{else}\ d \\
L\!:;
\end{array}
$$

Notice that this transformation does **not** preserve functional equivalence. It does not need to. It need only preserve the variable dependence relation. The outermost conditioned predicate is simply E, since the variable dependence upon the reference variable of E is all that is needed. Furthermore, the inner tests are merely present to create a conditional structure to replicate the optionality of the `switch`, but the dependence upon E has already been accounted for by the outermost E, and so the test is not repeated. This does not preserve functional equivalence, but does allow a correct computation of variable dependence, and does so more efficiently than the fully functional equivalent version of the nested conditional.

Moreover, the transformation to this nested conditional considerably simplifies the treatment of the problem of fall-through cases in `switch` statements. These fall through cases are a peculiar feature of C `switch` statements—that are often implemented as jump tables—in which each case is performed and execution can follow through to the cases below where there is no `break` statement to terminate the case. Also, `break` statements need not occur at the end of the code for the case, and there may be many.

It is therefore possible to create `switch` statements in which there is a nested conditional structure for some case and where this conditional has some paths containing `break` statements, while others simply cause execution to fall through to the lexically succeeding case.

A radical transformation that suggests itself would be to reverse the statements order of all statements in the program under inspection before applying *forward* dependence analysis in order to compute *backward* dependence. The result would be equivalent to variable dependence analysis as described here, but with the advantage of employing a tail recursive algorithm, which lends itself to optimisation. This possibility remains a topic for future work. However, the current implementation does contain a number of memoisation based performance enhancements, described in the next section.

7 Performance

The VADA system is designed to support the DAIMLERCHRYSLER Evolutionary Testing System, which is a test data generation system for unit level testing. Therefore the system will only have to scale to the size of a typical unit. This context of execution is kind to VADA. It means that a computation that takes a few minutes, or even more, can be forgiven if there is a reduction in the size of the search space of input variables. Since not all predicates of all programs will depend upon all input variables, the computational effort required by VADA will almost certainly be considered worthwhile.

The algorithmic complexity of the transformation phase is essentially linear as it involves several passes through the program, each of which is linear[4].

The complexity of the slicing and variable dependence analysis phase can vary dramatically, from linear in the best case to (potentially) exponential in the worst case. The worst case contains a sequence of loop carried dependencies (or 'backward assignment') in which each variable is assigned a value that depends upon the assignment that immediately follows it in the lexical order of

[4] As previously observed, the worst case for unfolding is exponential, but only in the depth of the call tree that grows logarithmically with program size and typically has a low value for unit level code.

the loop body. In the worst case, this 'backward assignment' is nested within a chain of nested loops, each of which contains killing assignments to the variables in the innermost loop and to the loop control variables. For such a program, the number of criteria that need to be considered by the innermost loop is exponential in the number of program variables (though not in the program size).

Clearly, such a highly artificial construction as the 'worst case' program is unlikely to be presented to VADA for analysis. However, the existence of such a program presents an upper bound on performance time and allowed an initial empirical evaluation of the performance of the system.

This analysis revealed that the performance in the presence of quite modest levels of loop nesting was unacceptably slow, so a memoisation feature was added to the variable dependence analysis phase. That is, a data structure of previously encountered slicing criteria is maintained for `while` loops. Where a criteria has previously been met, there is no point in re-computing the dependence. This creates a saving in computation time at the expense of memory space to record previously encountered criteria. However, it appears that the extra memory used by memoisation is more than offset by the reduction in call stack usage, as the memoisation reduces the average depth of recursion in the algorithm.

Interestingly, in this example of program analysis, the memoisation process can be further improved. Variable dependence is monotonic. That is, if $\mathrm{Vada}(S, V)$ denotes the set of variables whose initial values are determined by the set of variables V in the program S, then

$$U \subseteq V \Rightarrow \mathrm{Vada}(S, U) \subseteq \mathrm{Vada}(S, V)$$

This means that if a `while` loop is to be analysed with respect to a set of variables V, but a larger set has already been encountered in a previous computation, then the result of the previous computation can be used. Furthermore, variable dependence is distributive

$$\mathrm{Vada}(S, V \cup U) = \mathrm{Vada}(S, V) \cup \mathrm{Vada}(S, U)$$

This means that the results from several previous computations on subsets can be composed to form a partial answer to a current variable dependence question. These observations of monotonicity and distributivity are typically used to improve performance of variable dependence using monotone dataflow analysis frameworks [2]. Similar memoisation techniques are also used by Harrold and Ci [23] to improve slicing using dataflow based algorithms.

A memoisation algorithm was implemented that exploits monotonicity and distributivity of variable dependence and we obtained a significant improvement in worst case performance. The result of our initial study of performance are presented in next subsection.

7.1 Results

Timings taken on a AMD 1GHz based PC running Linux 2.4.8 with a memory bandwidth of circa 147MB/s (as measured by `hdparm`). Measurements were taken by averaging the sum of the user and system CPU time, as measured by the GNU `time` program, over three runs for each test case.

Figure 6 shows execution times for different numbers of worst-case backward assignments within five nested `while` loops, with and without memoisation.

Figure 7 shows execution times for different depths of `while`-loop nesting containing 50 worst-case backward assignments, with and without memoisation. The maximum depth of nesting that could be analysed without memoisation was restricted by memory constraints. In particular, the 32 bit architecture of the test machine imposed a maximum stack limit of 128MB (with SWI-Prolog) that was exhausted with depths of six or more (ten or more with fewer backward assignments). No attempt was made to find the corresponding limit with memoisation due to limits of testing time.

Figure 8 shows the execution times for forward assignments with no loops, where there is dependency on every assignment. There is no meaningful difference in the timings for the original and memoised versions of the system because the code that is responsible for managing memoisation is only invoked when analysing loops.

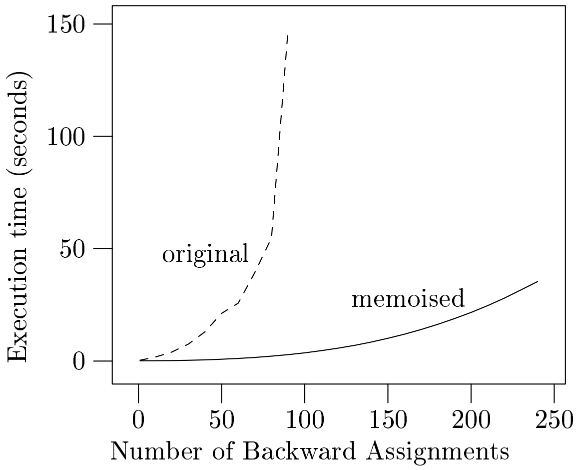

Figure 6. Timings for worst-case backward assignments within five nested `while` loops (timings without memoisation are shown by a dashed line)

One result deserves further discussion. Even with memoisation, the worst case performance of VADA displays an exponential component. The worst case per-

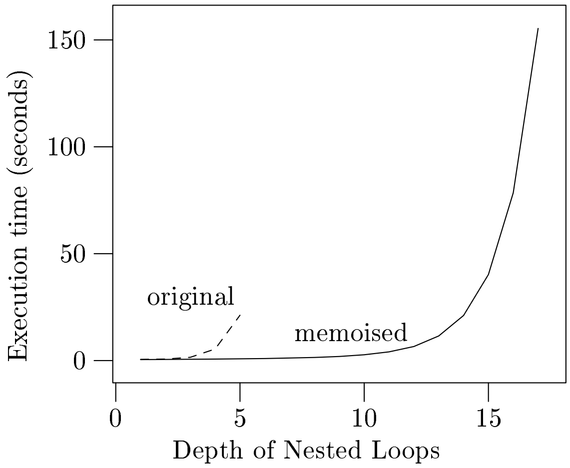

Figure 7. Timings for `while` loop nesting with 50 worst-case backward assignments (timings without memoisation are shown by a dashed line)

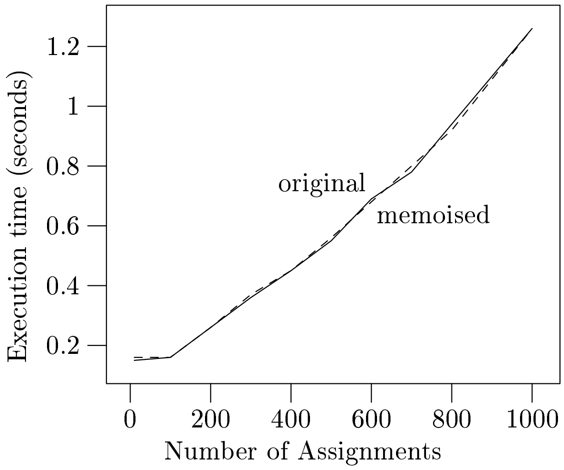

Figure 8. Timings for forward assignments with no loops (timings without memoisation are shown by a dashed line)

formance without memoisation is clearly far worse than the performance with memoisation, but the reader may be surprised that memoisation has not resulted in a linear performance. This is because the current version of memoisation, as implemented in VADA, uses a Prolog nested list data structure to store previously encountered criteria. This data structure becomes exponentially large as the degree of loop nesting increases. An improvement is planned using Prolog's built-in `assert` and `retract` mechanism that will allow the information about previously encountered criteria to be stored in a hash table, with near constant lookup time. The will further improve performance. The initial results from this trial implementation of memoisation are encouraging, and indicate that the hash-table version of VADA will have performance sufficient for use with the DAIMLERCHRYSLER Evolutionary Testing System.

8 Conclusion

This paper has presented an approach to computing variable dependence that requires program slicing for control dependence. The VADA system computes variable dependence for the C programming language. Constructing a slicer for the full range of C syntax and semantics is a large and complex undertaking. The VADA system adopts the approach of transforming the full C language to a more manageable core language, for which slicing and variables dependence are computed. A memoisation approach is used to improve the efficiency of the analysis phase and initial empirical results on the speed-up it produces have been presented.

A novel aspect of these transformations is that they are not meaning preserving. At least, they do not preserve the traditional functional equivalence relation that has remained the *sine qua non* of work on source-to-source transformation since the 1970s [12]. The VADA transformation engine need only preserve the variable dependence relation of a program.

References

[1] AGRAWAL, H., AND HORGAN, J. R. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, June 1990), pp. 246–256.

[2] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, techniques and tools*. Addison Wesley, 1986.

[3] BAXTER, I. D. Transformation systems: Domain-oriented component and implementation knowledge. In *Proceedings of the Ninth Workshop on Institutionalizing Software Reuse* (Austin, TX, USA, Jan. 1999).

[4] BENNETT, K. H. Do program transformations help reverse engineering? In *IEEE International Conference on Software Maintenance (ICSM'98)* (Bethesda, Maryland, USA, Nov. 1998), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 247–254.

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

[6] BINKLEY, D. W., AND GALLAGHER, K. B. Program slicing. In *Advances of Computing, Volume 43*, M. Zelkowitz, Ed. Academic Press, 1996, pp. 1–50.

[7] CANFORA, G., CIMITILE, A., AND DE LUCIA, A. Conditioned program slicing. In *Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier Science B. V., 1998, pp. 595–607.

[8] COUSOT, P., AND COUSOT, R. Systematic design of program transformation frameworks by abstract interpretation. *ACM SIGPLAN Notices 31*, 1 (Jan. 2002), 178–190.

[9] DANICIC, S. *Dataflow Minimal Slicing*. PhD thesis, University of North London, UK, School of Informatics, Apr. 1999.

[10] DANICIC, S., FOX, C., HARMAN, M., AND HIERONS, R. M. ConSIT: A conditioned program slicer. In *IEEE International Conference on Software Maintenance (ICSM'00)* (San Jose, California, USA, Oct. 2000), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 216–226.

[11] DANICIC, S., HARMAN, M., AND SIVAGURUNATHAN, Y. A parallel algorithm for static program slicing. *Information Processing Letters 56*, 6 (Dec. 1995), 307–313.

[12] DARLINGTON, J., AND BURSTALL, R. M. A system which automatically improves programs. *Acta Informatica 6* (1976), 41–60.

[13] DARLINGTON, J., AND BURSTALL, R. M. A tranformation system for developing recursive programs. *J. ACM 24*, 1 (1977), 44–67.

[14] DE LUCIA, A. Program slicing: Methods and applications. In 1^{st} *IEEE International Workshop on Source Code Analysis and Manipulation* (Florence, Italy, 2001), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 142–149.

[15] DERSHOWITZ, N., AND MANNA, Z. The evolution of programs: A system for automatic program modification. In *Conference Record of the Fourth Annual Symposium on Principles of Programming Languages* (1977), ACM SIGACT and SIGPLAN, ACM Press, pp. 144–154.

[16] FEATHER, M. S. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems 4*, 1 (Jan. 1982), 1–20.

[17] FERGUSON, R., AND KOREL, B. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology 5*, 1 (Jan. 1996), 63–86.

[18] FIELD, J., RAMALINGAM, G., AND TIP, F. Parametric program slicing. In 22^{nd} *ACM Symposium on Principles of Programming Languages* (San Francisco, CA, 1995), pp. 379–392.

[19] HARMAN, M., AND DANICIC, S. Amorphous program slicing. In 5^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'97)* (Dearborn, Michigan, USA, May 1997), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 70–79.

[20] HARMAN, M., AND HIERONS, R. M. An overview of program slicing. *Software Focus 2*, 3 (2001), 85–92.

[21] HARMAN, M., HIERONS, R. M., DANICIC, S., HOWROYD, J., AND FOX, C. Pre/post conditioned slicing. In *IEEE International Conference on Software Maintenance (ICSM'01)* (Florence, Italy, Nov. 2001), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 138–147.

[22] HARMAN, M., HU, L., ZHANG, X., AND MUNRO, M. Side-effect removal transformation. In 9^{th} *IEEE International Workshop on Program Comprehension (IWPC'01)* (Toronto, Canada, May 2001), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 310–319.

[23] HARROLD, M. J., AND CI, N. Reuse-driven interprocedural slicing. In *Proceedings of the 20th International Conference on Software Engineering* (Apr. 1998), IEEE Computer Society Press, pp. 74–83.

[24] HORWITZ, S., REPS, T., AND BINKLEY, D. W. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems 12*, 1 (1990), 26–61.

[25] JACKSON, D., AND ROLLINS, E. J. Chopping: A generalisation of slicing. Tech. Rep. CMU-CS-94-169, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1994.

[26] JONES, B., STHAMER, H.-H., AND EYRES, D. Automatic structural testing using genetic algorithms. *The Software Engineering Journal 11* (1996), 299–306.

[27] KOREL, B., AND LASKI, J. Dynamic program slicing. *Information Processing Letters 29*, 3 (Oct. 1988), 155–163.

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

[29] LANDIN, P. J. The next 700 programming languages. *Communications of the ACM 9*, 3 (Mar. 1966), 157–166.

[30] LAURENCE, M. R., DANICIC, S., HARMAN, M., HIERONS, R. M., AND HOWROYD, J. Equivalence of conservative, linear, free program schemas is decidable. *Theoretical Computer Science*. to appear.

[31] MICHAEL, C., MCGRAW, G., AND SCHATZ, M. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 12 (Dec. 2001), 1085–1110.

[32] MUELLER, F., AND WEGENER, J. A comparison of static analysis and evolutionary testing for the verification of timing constraints. In *4th IEEE Real-Time Technology and Applications Symposium (RTAS '98)* (Washington - Brussels - Tokyo, June 1998), IEEE, pp. 144–154.

[33] PARGAS, R. P., HARROLD, M. J., AND PECK, R. R. Test-data generation using genetic algorithms. *The Journal of Software Testing, Verification and Reliability 9* (1999), 263–282.

[34] PARTSCH, H. *The CIP Transformation System*. Springer, 1984, pp. 305–322. Peter Pepper (ed.).

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

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

[37] TRACEY, N., CLARK, J., AND MANDER, K. The way forward for unifying dynamic test-case generation: The optimisation-based approach. In *International Workshop on Dependable Computing and Its Applications (DCIA)* (January 1998), IFIP, pp. 169–180.

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

[39] WARD, M. Reverse engineering through formal transformation. *The Computer Journal 37*, 5 (1994).

[40] WARD, M. The formal approach to source code analysis and manipulation. In 1^{st} *IEEE International Workshop on Source Code Analysis and Manipulation* (Florence, Italy, 2001), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 185–193.

[41] WARD, M., CALLISS, F. W., AND MUNRO, M. The maintainer's assistant. In *Proceedings of the International Conference on Software Maintenance 1989* (1989), IEEE Computer Society Press, Los Alamitos, California, USA, p. 307.

[42] WEGENER, J., BARESEL, A., AND STHAMER, H. Evolutionary test environment for automatic structural testing. *Information and Software Technology Special Issue on Software Engineering using Metaheuristic Innovative Algorithms 43*, 14 (2001), 841–854.

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

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

[45] WEYUKER, E. J. *Program schemas with semantic restrictions*. PhD thesis, Rutgers University, New Brunswick, New Jersey, June 1977.