Abstract

Variable dependence is a source code analysis problem, related to slicing and chopping, in which the aim is to determine the set of variables that can affect the values of given program variables at specified points in the program. This paper describes an approach to solving the problem based on transforming program code into an intermediate language without preserving concrete semantics but keeping variable dependence relations intact. Interestingly the transformation phase need only preserve variable dependence, thereby admitting useful transformations that fail to preserve standard semantics (but do preserve variable dependence). The paper describes an implementation of a variable dependence analysis system called VADA, illustrating the application of non–meaning–preserving transformation, together with an empirical study of performance optimisation steps.

Key words: Variable Dependence Analysis, Slicing, Program Transformation.

1. Introduction

Variable dependence analysis is the problem of determining from a function, procedure of block of source code, the set of ‘input’ variables upon whose initial value a chosen computation depends. The choice of which variables form a valid set from which to draw inputs depends upon the context of the analysis. For instance in a function or procedure, the ‘input’ variables are typically taken to be the formal parameters of the function, while for a block of code they may simply be variables whose value is used before it is defined.

Variable dependence analysis is a dependence analysis that traces control and data dependences in a similar way to other program dependence analyses such as slicing [20,32] and chopping [22]. Like slicing, the criterion, is a pair (V,n), but instead of returning a set of statements that affect the value computed in variables V at line n, the algorithm returns the set of variables upon which the computation depends. Variable dependence has applications in program compre-
hension [5], where it can help to understand the impact of a change and in debugging and testing [17], where it can reduce the search space for unit test data generation.

This paper describes the design and implementation of the \textsc{V\alpha\beta\alpha} system, and in particular, its combination of transformation and analysis and its use of memoisation. \textsc{V\alpha\beta\alpha} is a system for variable dependence analysis of source code written in C. It is currently used by DaimlerChrysler as part of their approach to automated test data generation.

The \textsc{V\alpha\beta\alpha} tool was first introduced by Harman et al. [16]. The present paper is an extension of this previous workshop paper describing an augmented system that encompasses additional functionality and a wider target language. The present paper also extends the earlier workshop paper with additional implementation details and a formalisation of the analysis algorithm.

There are different kinds of variable dependence relationships. For example, a variable can be \textit{data dependent} or \textit{control dependent} on other variables. Data dependency arises from \textit{def, use} pairs of program points and can be described as follows. A variable $v$ defined at $n$ is said to be data dependent on another variable $v'$ at $n'$ if the latter is used in $n$ and there is an execution path from $n'$ to $n$ on which $v'$ is not modified. On the other hand, control dependency arises from program branching due to conditional statements or function calls. A variable $v$ defined at $n$ is said to be control dependent on another variable $v'$ at $n'$ if the execution or non-execution of $n$ depends on the truth value of the expression involving $v'$ at $n'$.

We explain the ideas of variable dependence relationships through the code fragment in Figure 1. This figure depicts a simple program with a single \texttt{while}-loop. In this example, the variable $x$ at the end of the program depends upon the initial values of the variables $z, y, b, a, x$. 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).

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. The size of the search space is defined by the number of input variables considered to be relevant to the computation of this predicate. Not all input variables contribute to the computation of every branch predicate. In cases where a predicate depends only on a subset of the input variables, it is wasteful to search the entire input space. \textsc{V\alpha\beta\alpha} supports search space reduction by finding the subsets of input variables relevant to the branches of a program. The approach adopted is similar to the chaining approach of Ferguson and Korel [15].

The principal contributions of the paper are as follows:
The paper introduces formal definition of the variable dependence algorithm. The paper presents description of the implementation of this approach in a tool called VADA. The paper introduces two techniques for performance improvement. A Memoization approach and a set theoretic optimization on variable dependence sets, based on properties of variable dependence. The paper also presents an empirical study of their effects.

The core language allows computation of variable dependence for the core language, while its results are applicable to the full language. As will be seen, the transformation into the core language also includes novel transformation steps, available only for this application, because they need preserve only variable dependence, not traditional functional meaning.

The rest of this paper is organised as follows. Section 2 shows the relationship between slicing and variable dependence. Section 3 describes the overall architecture and the implementation of the VADA system. Section 2 introduces the core language used by the slicing engine at its heart. Sections 2 and 3.1 describe the transformation and variable dependence analysis phases of the system in more detail and Section 4 presents some initial empirical results concerning the performance of the analysis phase. Section 5 gives a worked example. Section 6 deals with related work. Section 7 ends the paper with some concluding remarks.

2. Variable Dependence Analysis Approach

In this section, our approach to variable dependence analysis is described. The approach is based on rewriting program code in an intermediate core language using a subset of the C programming language. The transformation algorithm simplifies the analysis by removing semantics information irrelevant to dependency relationships among program variables. The analysis algorithm is expressed as a transformation mapping program statements combined with their variables into program statements.

2.1. The Core Language

The core language consists of a simple subset of intraprocedural C. The subset does not restrict expression notation, but only supports a limited set of statement constructs. Procedures as such are also not supported directly. More complex statement constructions and non-recursive procedures are instead interpreted by translation into the core language. The syntax of the core language is defined in Figure 2.

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. Unfolding is known to have worst case exponential growth in the depth of call tree. However, this does not lead to problems in practice, because

(i) The algorithmic complexity of function unfolding is exponential in the depth of the call tree and since the depth of the call tree grows logarithmically with its size, unfolding functions does not lead to exponential explosion.

(ii) At the unit level, program size is normally relatively small.

(iii) Many applications in the industry, such as those in embedded systems tend to have shallow flat call trees.
The core language was chosen to facilitate easy translation from full C, whilst allowing the control flow relationships of the full language to be preserved by translation. The while loop and if statement are retained as the paradigms of selection and repetition. All conditional and iteration constructs can be converted into these core constructs in a way that preserves the control dependence relationships between variables. A smaller core language would have been possible \[30\], 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. It is also easier to manually verify the individual translation and dependence rules than it is to check rules that compute the dependencies directly from more esoteric program constructs.

2.2. Program Transformation

The transformation algorithm transforms C programs into the core language, the syntax of which was defined in Figure 2. 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.
2.2.1. Step 1 (dependence criteria)

This step inserts assignment statements to pseudo variables to capture the variable dependence criterion specified by the user. 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 \( \langle \{v_1, v_2, \ldots, v_n\}, p \rangle \) where \( v_1, \ldots, v_n \) are program variables, and \( p \) is a point of interest, an assignment \( \pi = \pi + v_1 + \ldots + v_n \) is inserted at the point of interest, where \( \pi \) is a pseudo variable not otherwise used in the program.

The fact that the expression assigned to this pseudo variable involves additions is an arbitrary choice that just ensures that the variables \( v_1, \ldots, v_n \) are referenced. The pseudo variable itself is also included as a referenced variable in order to correctly compute dependence on \( v_1, \ldots, v_n \) when the point of interest lies inside a loop.

The point of interest \( p \) is given by identifying a particular expression in the program, rather than a statement, for example, expressions are given unique identifiers in the AST representation of the program under analysis. Expression identifiers can be used to define dependence criteria, in which case, all of the variables mentioned in that expression are included in the criteria.

2.2.2. Step 2 (globalisation)

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.

2.2.3. Step 3 (symbol table)

In this step, a mapping is created from procedure and function names to their bodies. This is a pre-requisite for Step 4. There is a sense in which it is not really a transformation step itself because it does not affect the program, but it is required in order to unfold the procedures by replacing procedure calls with the corresponding program code.

2.2.4. Step 4 (procedure and function unfolding)

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.

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 analysis program needs 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 in taking into account the calling context.

Function calls are transformed to procedure calls with an additional fresh global variable to communicate the result to the caller. While this may not be good programming practice, the code will be seen only by an analysis tool and not by a human. Furthermore, the all-important dependence relation remains invariant.
2.2.5. **Step 5 (second globalisation)**

The fifth step 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.

2.2.6. **Step 6 (statement transformations)**

This final step performs the 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 \( \langle \text{identifier} \rangle = \langle \text{expression} \rangle; \). 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 3.1 and memoisation, as explained in Section 4. It also has a positive benefit on testing and verification of the approach, 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. Additional labels are introduced in order to capture the targets of these goto statements. This allows all these three forward jumps 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.

The next subsection gives an example of such a transformation, where a C statement with a relatively complex semantics is transformed into the core language in a way that preserves variable dependencies but which avoids the additional complications that would be required to preserve full functional equivalence.

2.2.7. **Example: transforming switch statements**

The transformation function \( R^{(label)}(\text{statement}) \) produces a statement or statement sequence \( \langle \text{statement} \rangle' \) from a statement or statement sequence \( \langle \text{statement} \rangle \). \( \langle \text{statement} \rangle' \) is identical to \( \langle \text{statement} \rangle \) except that all occurrences of the break statement have been replaced by the statement goto \( \langle \text{label} \rangle; \). The transformation of a switch statement converts the switch into a nested conditional, using \( R^{(label)}(\text{statement}) \) to replace break statements with forward jumps that exit the nested structure. More formally, the switch statement of Figure 3 is transformed into the nested conditional statement of Figure 4.

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 \( \langle \text{expression} \rangle \), since the variable dependence upon the reference variable of \( \langle \text{expression} \rangle \) 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 \( \langle \text{expression} \rangle \) has already been
Fig. 3. A generic switch statement

```
switch ((expression)) {
    case c_1 : (statement)_1
    :
    case c_n : (statement)_n
    default : (statement)_d
}
```

Fig. 4. A transformed switch statement

```
if ((expression)) {
    if (1) {
        if (1) {
            :.
            if (1) {
                if (1) {
                    R(label)(statement)_1
                    R(label)(statement)_2
                .
                R(label)(statement)_{n-1}
                R(label)(statement)_n
            } else (statement)_d
            (label) ::

```

accounted for by the outermost (expression), 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 functionally 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 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.

It is also worth noting that break statements need not occur merely at the end of a case, and there may be many breaks within the program code handling each case. For example, it is 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.

The transformational flexibility available to the initial stage is considerable, since only variable dependence need be preserved. For example, a more radical transformation which suggests itself would be to reverse the statements order of all the 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.

2.3. Variable Dependence Algorithm

In this section, a variable dependence analysis algorithm is presented as a function on core language statements. According to the core language syntax defined in Section 2.1, there are effectively five different types of statements: assignment, conditional (if), loop (while), sequence (of statements), and goto. The function takes a set of identifiers (variables) together with a statement as parameters and delivers a set of identifiers of the statement’s expression that are control and data dependent on the parameter variables. The set of parameter variables with which the analysis starts is chosen to be the set of identifiers found in a program expression or a statement and is called the dependence criterion. As the program is analysed in sequential order, the starting set of variables representing the dependence criterion changes depending on the program control and data flow.

Let \( S \) denote the set of program statements expressed in the core language, and \( V \) the set of program identifiers. A \( vda \) function is a map that takes as input a program core language statement and a set of identifiers and produces as output a set of variables that are dependent on the given parameter set.

\[
vda : V \times S \rightarrow V
\]

In the following, \([s] \) where \( s \in S \) denotes “approximate” semantics of \( s \) resulting from the transformation of original program statement with “concrete” semantics into the core language-as described in Section 2.2 on statement transformation.

2.3.1. An assignment statement: \([v = E;]\)

There are two cases depending on whether the assigned variable (\( v \)) is of interest, i.e., belongs to the criterion set of variables \( V \) as it reaches the statement, or not. In the case where \( v \notin V \), the statement is not considered and the output set is identical to the original set \( V \). Otherwise, a set containing all input variables (apart from \( v \)) together with all the variables referenced in expression \( E \) is delivered.

In terms of the \( vda \) transformation, the two cases can be more formally expressed as follows. Here, \( Ref(E) \) denotes all variables referenced in expression \( E \) [1].

\[
vda (V, [v = E]) = V, \quad \text{if } v \notin V
\]

\[
vda (V, [v = E]) = (V - \{v\}) \cup Ref(E), \quad \text{if } v \in V
\]

2.3.2. An if-statement: \([ if \ P \ then \ S_1 \ else \ S_2 ]\)

Here, \( S_1 \) and \( S_2 \) represent sets of statements and there are two cases to be considered.

(1) the input variables are independent of \( S_1 \) and \( S_2 \), or, in other words, the latter do not affect the input variables. In this case, the input set of variables \( V \) is delivered.
\[
\text{vda} ([\text{if } P \text{ then } S_1 \text{ else } S_2], V) = V,
\]

where \(\text{vda}(S_1, V) = V\) and \(\text{vda}(S_2, V) = V\)

(2) statements in \(S_1\) and in \(S_2\) do affect the input variables. In this case, the variables of the such statements define, in combination with variables referenced in the predicate \(P\), the output set of variables. We note that the set of variables referenced in the predicate, \(\text{Ref}(P)\), becomes relevant (due to control dependency) only if subsets of \(S_1\) or \(S_2\) that affect the input variables are not empty.

In the following, \(V_1\) and \(V_2\) represent, respectively, sets of variables in those statements of \(S_1\) and \(S_2\) that affect the input variables, or, in other words, on which the input variables are dependent.

\[
\text{vda} ([\text{if } P \text{ then } S_1 \text{ else } S_2], V) = V_1 \cup V_2 \cup \text{Ref}(P),
\]

where \(\text{vda}(S_1, V) = V_1\) and \(\text{vda}(S_2, V) = V_2\)

An if-statements without “else”, that is, with empty \(S_2 = [\text{ ]}\), represents a special case and can be treated in the same way.

2.3.3. A sequence of statements: \([S_1; S_2]\)

We assume that the statement are executed in the given order from right to left, i.e. \(S_2\) followed by \(S_1\). If a sequence of sets of statements have no effect on the input set of variables \(V\), the original set \(V\) is returned. Otherwise, the set of variable dependencies resulting from executing \(S_2\) is used as variable criterion in combination with \(S_1\).

\[
\text{vda} ([S_1; S_2], V) = V',
\]

where \(\text{vda}(S_2, V) = V_2\) and \(\text{vda}(S_1, V_2) = V'\)

2.3.4. A while loop: \([\text{while}(P) \text{ do } S]\)

In the case where the loop body \(S\) does not affect the criterion input variable set, the input set of variables \(V\) is delivered.

\[
\text{vda} ([\text{while}(P) \text{ do } S], V) = V, \text{ where } \text{vda}(S, V) = V
\]

In the other case where \(S\) affects \(V\), \(\text{vda}\) computes the set of variables corresponding to the fixed point. This is the smallest set of variables that contains \(V\) and that remains invariant after the loop body is completed. The fixed point corresponds to the condition \(V'_n \subseteq V'_{n+1}\) in the following expression. The returned set of criterion variables combines the input variables set, the set corresponding to the fixed point and the variables referenced in the predicate \(P\).

In the following, \(S'_i\) represents the subset of statements of the loops body \((S)\) that affects the current set of criterion variables after the \(i^{th}\) iteration of the loop.

9
\[
vda (\llbracket \text{while}(P) \ \text{do} \ S \rrbracket, V) = V',
\]
where \(vda(S, V) = V'_1\) and
\[
vda(S'_1, V'_1 \cup \text{Ref}(P) \cup V) = V'_2\quad \text{and}
vda(S'_2, V'_2) = (S'_3, V'_3) \quad \text{and}
\]
\[
\vdots
\]
\[
vda(S''_n, V''_n) = V''_{n+1}
\]
with \(V'_n \subseteq V'_{n+1}\) and \(V'' = V'_n \cup \text{Ref}(P) \cup V\)

### 2.3.5. A goto statement: \(\llbracket \text{goto} \ L \rrbracket\)

For a goto statement with label \(L\), \(vda\) computes the criteria to which \(L\) is mapped and updates the criteria as new labels are encountered. Given the mapping information, \(vda\) can proceed to calculate variable dependencies.

### 3. The VADA Implementation of Variable Dependence Analysis Approach

\textit{VADA} is a Prolog implementation of the variable dependence analysis approach described in the previous sections. Precisely, \textit{VADA} uses SWI-Prolog \(^1\) to implement a syntax-directed variable dependence algorithm, derived from the parallel slicing algorithm of Danicic and Harman [9]. The reasons for choosing Prolog are manifold. Prolog itself is a versatile meta-programming language. Related to this, recursive pattern matching rules with mappings to some interpretation (in this case dependence) can be expressed in a fairly natural way. This feature helps with rapid development, modulo the issue of having to thread values through arguments.

Other languages, e.g., Haskell and ML, and other implementations of Prolog also have some of these features, but essentially the most important feature in this case is SWI-Prolog’s SGML/XML parser. It is one of the fastest, most versatile and compliant SGML/XML parsers. It can parse with or without a DTD, give informative error messages about the DTD (which actually helped identify bugs in the third party C to XML parser) and generate a DTD from a sample file. It also has a hybrid event-based/tree-based parser, so that one can set an event that triggers a tree-based parse of the matching entity. As the SWI-Prolog parser output is a Prolog term, it naturally feeds into the pattern-matching/tree-based parser paradigm that Prolog supports. SWI-Prolog is appropriate for that kind of analysis problem \textit{VADA} addresses. SWI-Prolog also supports the creation of stand-alone executables under Linux and MS–Windows from a common source.

The overall implementation is split into three main components: a transformer, an analyser and a user interface. The transformer converts the AST representation of the source code, expressed as an XML tree, into the core language. The core language representation becomes the subject of the analysis phase. This follows the approach first suggested by Landin [24]. The interface provides means of input and output, which allows users to manipulate the system, to define the criteria for analysis and to receive analysis results.

The transformation algorithm preserves the variable dependence relations though not necessarily the full semantics of the original program. The transformation serves to simplify the role

---

\(^1\) SWI-Prolog is available under the GNU Public Licence from http://www.swi.psy.uva.nl/projects/SWI-Prolog/.
of the analyser. The overall system accepts ANSI C. However, some language features such as recursion and backward goto statements are not currently supported.

3.1. The Variable Dependence Analyser

The Prolog rules used by VAda are slicing rules for the the core language that yield variable dependence information as a by-product. The rules are attractively simple to define. The essence of the slicing engine, in Prolog, is presented in Figure 5. It largely consists of pattern matching and rewriting rules for the core language constructs. The auxiliary predicates (for which detail is not provided) are described in English in Figure 6.

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. 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.

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 [1], but it remains open as to whether unreachable code can be determined at the level of abstraction used for computing variable dependence [8,25,33].

The parsing details are left abstract in the exposition. In the implementation, the pattern matching and rewriting is actually expressed in terms of selectors and constructors operating on Prolog representations of an XML encoding of the relevant constructs. To aid comprehension, and avoid cluttering the details of the algorithm, the rules are given in the form of quasi quoted [ ... ] C constructs in the figure. These quotes (or ‘syntactic brackets’) are typically used to distinguish the language under analysis from the language used to perform the analysis [28].

The predicate slice in Figure 5 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

\[
slice(MapIn,MapOut,\]

\[
\text{StatementIn,CritIn,}\]

\[
\text{StatementOut,CritOut})\]

\[
takes as input the representation of a C program statement StatementIn and a criterion (a list of variables of interest) CritIn. It produces as output a slice StatementOut and an output criterion (a list of variables upon which CritIn depends), CritOut. The slice StatementOut is the end slice [23] of the statement StatementIn with respect to the list of variables CritIn. CritOut is the list of variables whose initial values can, in principle at least, determine the values of the variables in CritIn during the course of some execution of the object language program code StatementIn.

In addition, slice also takes, as input, a mapping from labels (goto targets) to slicing criteria (lists of variables), MapIn and produces an updated mapping, MapOut as an output. This mapping
slice(MapIn,MapIn,[\[I=E;\]],[CritIn],[\[I=E;\]],[CritOut]) :-
    intersection(CritIn,[I],Intersection),
    not(Intersection = []),
    collect_refs([E],VarNamesR),
    subtract(CritIn,[I],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]),
    slice(MapIn,ElseMapOut,[\[C;\]],[CritIn],[\[C';\]],[CritElse]),
    (not(empty_statement([E]));not(empty_statement([E']))),
    collect_refs([E],PredRefs),
    big_union([CritThen,CritElse,PredRefs],CritOut).

slice(MapIn,MapOut,[\[if(E)C1 \text{ else } C2;\]],[CritIn],[\[if(E)C'1 \text{ else } C'2;\]],[CritOut]) :-
    slice(MapIn,ThenMapOut,[\[C1;\]],[CritIn],[\[C'1;\]],[CritThen]),
    slice(MapIn,ElseMapOut,[\[C2;\]],[CritIn],[\[C'2;\]],[CritElse]),
    (not(empty_statement([E]));not(empty_statement([E']))),
    collect_refs([E],PredRefs),
    union(ThenMapOut,ElseMapOut,MapOut),
    big_union([CritThen,CritElse,PredRefs],CritOut).

slice(MapIn,MapIn,[\[while(E)C;\]],[CritIn],[\[\]],[CritIn]).

slice(MapIn,MapOut,[\{C1 ... C_{n-1} C_n\}],[CritIn],[\{C'1 ... C'_{n-1} C'_n\}],[CritOut]) :-
    slice(MapIn,MapInternal,[\[C_n;\]],[CritIn],[\[C'_n;\]],[CritInternal]),
    slice(MapInternal,MapOut,[\[C1 ... C_{n-1}\}],[CritInternal],[\[C'1 ... 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,PrevButOneCrit,PrevCrit,Body,CritIn,PrevCrit) :-
    subset(PrevCrit,PrevButOneCrit).

find_fixpoint(MapIn,MapIn,PrevButOneCrit,PrevCrit,Body,CritIn,FinalCrit) :-
    big_union([PrevCrit,PrevRefs,CritIn],NewCritIn),
    slice(MapIn,MapOut',Body,NewCritIn),
    find_fixpoint(MapOut',MapOut',PrevCrit,Body,CritIn,FinalCrit).

Fig. 5. Core Language Slicing Algorithm in Prolog Pseudo Code
<table>
<thead>
<tr>
<th>Prolog Predicate</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>big_union</td>
<td>Forms the distributed union of a set of sets</td>
</tr>
<tr>
<td>collect refs</td>
<td>Determines the referenced variables of an expression</td>
</tr>
<tr>
<td>empty statement</td>
<td>Determines whether a statement contains only dead code</td>
</tr>
<tr>
<td>and empty structures</td>
<td></td>
</tr>
<tr>
<td>lookupMap</td>
<td>Find the criteria to which a label is mapped</td>
</tr>
<tr>
<td>updateMap</td>
<td>Update a map to record a new mapping from a Label to a criterion</td>
</tr>
</tbody>
</table>

Fig. 6. Details of additional Prolog predicates required by the slicer/variable dependence analyser

information is required in order to slice, and hence compute variable dependency information for program code involving `goto` statements. The mapping `MapIn` records the slicing criterion that was computed for any 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 `CritOut` for `while` loops) the mapping is also updated for previously encountered labels that are re-encountered as the iteration to the fixedpoint proceeds. The updated mapping is given in `MapOut`.

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 [9,32]. Each rule is processed according to Prolog’s ordered pattern-matching rules. The rule that keeps a C statement in the slice is considered first, using the default assumption that the statement may contain information that has an impact on the variables in question. If it turns out that this assumption is incorrect, and the C statement has no impact on the values of the variables in the slicing/dependency criteria, then the rule fails. The entire C statement is then removed from the slice by the clause that follows, and the dependency criteria is left unchanged.

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, giving 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).

The parser treats C program labels as statements. The rules for labels and `goto` statements, merely record and restore, respectively, the criterion associated with the label mentioned in the statement.

### 3.2. Architecture and Modules

Figure 7 shows a view of the VADA system as a processing machine which expects three input parameters and gives one output. The input parameters represent, respectively, an upper time limit for the analysis execution time, the source code and the criterion according to which the code should be analysed. The timer component is not shown in the figure, because it behaves an external interrupt that merely controls the processing time. The output represents the analysis
result expressed in XML format.

Figure 8 shows the modules of the system and their dependencies. In this figure, an arrow denotes module dependency. An arrow from module A to module B means that A is dependent on B, or, in other words, it uses the predicates of module B for its implementation. Prolog modules are used with use_module declarations which import public predicates of the modules into the imported program. The modules vadaXTimer and vadaCTimer provide the command-line interface for the stand-alone VADA system. Both modules provide a predicate vadaTimer expecting four parameters.

The modules can be classified into the following categories:

**User interface** User interface modules implement the SWI-Prolog mechanism for building program executables defining the program entry point. This is achieved by vadaXTimer and vadaCTimer. These modules allow the user to start VADA given the appropriate input parameters. For the output, there are modules which transform temporary/final results of core language transformation or analysis from internal Prolog representation into XML format and save them. The modules fileutils and ppsgml provide predicates for writing and saving analysis results in XML format. More on the user interface in the following section 3.4.

**Parser interface** Interface to the SWI-Parser is needed for loading and parsing the AST representation of the source code and the criterion delivered as XML files into Prolog SGML structures. This is realised through utils.
Transformation  Transforming the XML-encoded AST representation of the source code together with the criterion into the core language. This is achieved by trans.

Analysis  Analysing the transformed code for variable dependencies. This is realised through slice, defined and used.

Control flow  Controlling the program flow between various modules. This task is taken by glue. The glue module plays the role of a bridge between transformation, analysis, loading and parsing files as Prolog SGML terms, modules receiving user input and those generating output analysis. Most of the Vαα modules are used by glue. The predicates of these modules are called to load the AST representation of the source file and the criterion, to perform transformation and analysis and to deliver the results of the variable dependence analysis in the (fourth) output parameter file in XML format.

Configuration  The module options is used to set global configuration parameters.

Debugging  Modules developed for interactive use with the SWI-Prolog environment and meant to serve development and debugging purposes. These modules are not shown in Figure 8.

3.3. Variable Dependence Relationships

In addition to calculating the set of dependent variables according to specified criteria defined locally in terms of variables at a program point, Vαα also computes global dependency properties among all program variables and formal parameters of any function selected for analysis, or for the entire program.

All program variables, including global variables, local variables and formal parameters are divided into five different variable dependency sets or relationships. These relationships are: used, defined, definitely defined, needed, and controlling. In the following, the Control Flow Graph (CFG) program representation is used to define the variable dependency relationships.

Used  A set of used variables is one whose elements may be referenced before or after a variable is assigned along some paths of the CFG from start node to the stop node.

Defined  A set of defined variables is one whose elements occur on the left-hand side of an assignment node. Such variables are sometimes called “possibly defined” or “possibly assigned”, because they are not assigned on every path of the CFG from start node to the stop node.

Definitely defined  A set of definitely defined variables is one whose elements are assigned (appear on the left-hand side of an assignment node) on every path of the CFG from start node to the stop node.

Needed  A set of needed variables (for a variable $v$) is one whose elements’ initial values potentially affect the final value of the variable they are needed for (i.e., $v$) along some paths of the CFG.

Controlling  A set of controlling variables is one whose elements are needed for some variable $v$ and $v$ appears in a predicate node.

There are inclusion relationships between the variable sets just defined. If used denotes the set of used variables, defined the set of defined variables, needed the set of needed variables, defined$_{def}$ the set of definitely defined variables and controlling the set of controlling variables, we note that,
Fig. 8. Vada modules
Next, we give some explanation examples on the variable dependence relationships as computed by \textsc{Vada}.

(i) For an assignment statement of the form: \(x = y + z\),
the variable \(x\) is \textit{definitely defined} and consequently also \textit{defined}. The variables \(y\) and \(z\) are \textit{used} and \textit{needed}.

(ii) For an if-statement of the form: if \((p > q)\);
the variables \(p\) and \(q\) are \textit{controlling} and consequently \textit{used} but neither \textit{needed} nor \textit{defined}.

(iii) For an if-else-statement of the form: if \((p > q)\) \(x = 1\); else \{\(y = 1;\) \(x = 2\);\},
the variable \(x\) is \textit{definitely defined} and consequently \textit{defined}. The variable \(y\) is \textit{defined} and \textit{needed}. The variables \(p\) and \(q\) are \textit{needed}, \textit{controlling} and consequently \textit{used}.

(iv) For a sequence of program statements of the form: \(x = 1;\) \(y = 1;\) if \((x > y)\) \(z = y;\),
the variables \(x\) and \(y\) are \textit{definitely defined} and consequently \textit{defined}. The variable \(z\) is \textit{defined} and \textit{needed}.

(v) For an if-statement of the form: if \((p > q)\) \(z = 1;\),
the variables \(p\) and \(q\) are \textit{controlling} and consequently \textit{used}, and \textit{needed}. The variable \(z\) is \textit{defined} and \textit{needed} but not \textit{used}. \(z\) is \textit{needed} in order to define the final value of itself, since it may remain unchanged.

The last example illustrates why \textit{needed} \(\not\subseteq\) \textit{used}.

### 3.4. The User Interface

In this section the user interface which encompasses \texttt{vadaXTimer} and \texttt{vadaCTimer} is described in details. The difference between these modules lies in the representation of the program submitted for analysis. The \texttt{vadaXTimer} predicate expects an XML-encoded AST representation whereas the \texttt{vadaCTimer} predicate the source code itself. Internally, \texttt{Vada} transforms the source code into XML-encoded AST representation before proceeding with the analysis. The stand-alone \texttt{Vada} system expects four parameters shown in Figure~9. In this figure, the entries in the first column refer to the roles of the parameters, followed by brief descriptions.

\texttt{Vada} uses SWI-Prolog’s capability of handling software interrupts (signals) for implementing the timer functionality. Special cases arise from setting the timer parameter. A zero (integer) value switches the timer off; for any other positive value the timer demon is on. In cases where the parameter value is non-zero but less than the time required for completing the analysis, pro-
cessing the code is stopped without producing final results. In cases where the timer is off, VADA stops processing the code only after the final result is ready and saved in a file named through the fourth parameter.

The second parameter specifies the file of the code to be analysed. The XML-encoded AST representation used internally by VADA is generated by a proprietary parser licensed to Daimler-Chrysler.

The third parameter specifies the analysis criteria. In this file all the functions of a program and all, or a subset of their statements can be specified as part of the analysis criteria. The XML structure of the criteria is shown in Figure 10. A DTD file is not required. For functions, there is a function-tag with a name attribute and the statements to be analysed are specified as children. In Figure 10, the function names are set arbitrarily. The values of the expression-tag attributes exprref refer to statements as they appear in the XML-encoded AST representation of the source code. Again, the expressions used to represent the attributes’ values reflect the values’ structure, where repetitive sequences of letters such as nnn denote natural numbers. For example, if branch coverage for a certain function is required, the exprref entries correspond to all conditional and loop statements that appear in the code of the function.

VADA requires a criterion file before the analysis process can start. It is not required that all the functions or their statements be specified. However, at least one function must be named. Figure 11 shows the simplest criteria possible where only a function named foo is specified. Here, foo is assumed to be part of the program to be analysed. If the given name does not match any of the program functions, VADA reports an error informing the user that the function has not been found.

A regular dependency criteria is always defined with respect to a program function. The five dependency relationships of Section 3.3 are always computed for every program function specified in the criterion. Program functions can also be selected without mention of any specific variables and expressions (program points), in which case only the dependency relationships are computed for the selected functions.

If all four parameters are correctly given and the timer value is sufficiently long or set to (integer) zero, VADA generates an analysis output and saves the results in the file whose name is specified by the fourth parameter. In this result, variables dependency relationships of all the functions specified in the analysis criteria are listed. VADA extends the set of variable dependency relationships including Defined and Used by adding three more relationships: Definitely defined, Needed and Controlling). In Section ?? these relationships are defined and discussed in more detail. In addition, variables’ dependencies of statements specified in the criteria are also listed. If no statements are specified, only dependency relationships of all the variables of specified functions are listed. Figure 12 depicts the XML structure of the analysis results. The figure shows that VADA, in addition to input variables, also includes local variables in the list attaching a local_n to these variables as prefix. Here, n denotes a number. A variable-tag has two attributes specifying its name and its idref as it appears in the XML-encoded AST representation of the source code.

4. Empirical Study

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 means that a computation that takes a few minutes,
or even more, will be acceptable when 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. An empirical study by Binkley and Harman [4] revealed that predicates do not depend on all formal parameters and also that as the number of formals increases, the portion on which a typical predicate depends decreases.

4.1. Performance and Memoisation

The algorithmic complexity of the transformation phase is essentially linear because it involves several passes through the program, each of which is linear. 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 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 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 forms the basis for empirical evaluation of the performance of the system.

This analysis revealed that the performance could potentially be unacceptably slow, so a memoisation feature is 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.

Memoisation is very applicable as a performance enhancement technique because criteria are repeatedly re-encountered during the fixed point computation for loops. The memoisation algorithm used by Vada is not merely a case of recording previously encountered criteria in order to reuse them later. The algorithm exploits several useful properties of variable dependence. Variable dependence is monotonic. That is, if Vada(S, V) denotes the set of variables whose initial values are determined by the set of variables V in the program S, then

\[ U \subseteq V \Rightarrow Vada(S, U) \subseteq Vada(S, V) \]

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

\[ Vada(S, V \cup U) = Vada(S, V) \cup Vada(S, U) \]

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 often used to improve performance of variable dependence using monotone dataflow analysis frameworks [1]. Similar memoisation techniques are also used by Harrold and Ci [19] 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. An empirical study of performance improvement is presented in next subsection.

4.2. 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 13 shows execution times for different numbers of worst-case backward assignments within five nested while loops, with and without memoisation.

Figure 14 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).

Figure 15 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.

![Graph showing execution times for forward assignments with no loops.](image1)

Fig. 13. Timings for worst-case backward assignments within five nested **while** loops (timings without memoisation are shown by a dashed line)

![Graph showing execution times for **while** loop nesting with 50 worst-case backward assignments.](image2)

Fig. 14. Timings for **while** loop nesting with 50 worst-case backward assignments (timings without memoisation are shown by a dashed line)
5. Related Work

The problem of variable dependence is closely related to program slicing [12,18,32] and chopping [21] because it involves the traversal of transitive data and control dependence relations. In our implementation, the variable dependences for a program that is expressed in the whole language are computed by way of a slicing algorithm that operates on the program following its transformation into the core language.

Danicic [8] showed 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 Program Dependence Graph (PDG) [26] typically denote expressions (either the boolean expressions of conditional statements or the arithmetic expressions of assignments).

Whereby program transformations are expected to preserve some form of functional equivalence [2,3,11,27,29,31], transformations proposed in this work are not meaning preserving in the traditional sense. Although non-meaning preserving transformations have been considered for program modification in corrective and adaptive maintenance [13,14], the use of non-meaning preserving 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 semantic transformation of Section 2.2 that accompany the syntactic transformation into the core language defines an abstract interpretation [6,7] with an abstract domain which preserves the property of variable dependence. Here, the concrete semantic domain is approximated at the expense of simplifying the variable dependence analysis. This is similar to a non-relational interval domain where the interval analysis is simplified and thus made fast by overlooking the
relationships between variables (e.g. equality). The abstract interpretation is consistent with the concrete semantics of the original program in maintaining variable dependency since control flow is preserved in the transformation.

6. Conclusion

The \textsc{V\textcopyright A\textcopyright D\textcopyright A} system computes variable dependence for the C programming language. The \textsc{V\textcopyright A\textcopyright D\textcopyright A} system adopts the approach of transforming the full C language to a more manageable core language, for which variable dependence is 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 \textit{sine qua non} of work on source-to-source transformation since the 1970s [10]. The \textsc{V\textcopyright A\textcopyright D\textcopyright A} transformation engine need only preserve the variable dependence relation of a program.

The paper describes the transformation algorithm and formalises the analysis algorithm, showing how it has a natural expression in Prolog pseudo code. The paper describes an implementation of this approach and evaluates the performance improvement due to memoisation.

References


