Dependency Analysis of Ada Programs

Janusz Laski
SofTools, Inc.
3024 Longview
Rochester Hills MI, 48307
(248) 853-7602
laski@oakland.edu

William Stanley
SofTools, Inc.
3024 Longview
Rochester Hills MI, 48307
(248) 853-7602
wfstanle@oakland.edu

Jim Hurst
SofTools, Inc.
3024 Longview
Rochester Hills MI, 48307
(248) 853-7602
jshurst@oakland.edu

1. ABSTRACT

The working hypothesis of this paper has been the belief that Software Testing and Analysis (STA) methods should be integrated around a common conceptual framework. An analysis of two potential candidates for such a framework, Program Dependencies and Information Flow relations, shows that the ideal framework should posses the mathematical elegance of the flow relations and the generality of program dependencies. However, program dependencies have originally been formulated for compiler optimization and their uncritical use in software engineering is lacking. Therefore, modifications to the original dependencies have been proposed in this paper. They include partial vs. total definitions of Ada arrays, new concept of reaching definitions, potential Vs guaranteed dependencies, interprocedural dependencies, and an explanation feature that helps the user understand the reasons for the generated reports. It has been shown how the modified model can support descriptive and proscriptive (e.g. anomalies) queries about the program and, due to the clear separation of control flow from data flow and the lack of language restrictions, they are potentially applicable to a wider class of STA methods, including dynamic (execution-based) analysis. Also, Path Analysis, a novel method for the identification of dependencies along individual program paths has been proposed. It has been shown that path analysis offers a more accurate model for procedure calls, allows one to detect otherwise undetectable data flow anomalies and can serve as a vehicle for the analysis of error creation and propagation in testing and debugging.

1.1 Keywords

Program dependencies, information flow, static analysis, path analysis, Ada, program anomalies.

2. INTRODUCTION

Recent progress in software development methods, especially those based on mathematical modeling (e.g. the Vienna Development Method, [JONE90]) will eventually render programs that are "almost correct." Ironically, this success of software design methodology makes program testing and analysis even more challenging. Clearly, while it is relatively easy to detect faults in a badly designed code, it is much more difficult to detect hidden, residual faults. This poses a new challenge for *Software Testing and Analysis (STA)* methods, which apply to those phases in the software development cycle where a compilable code is available. Ideally, STA should offer support in *software verification*, i.e. reasoning about its correctness. Unfortunately, the support offered by commercially available tools is often of clerical (e.g. “Capture and Play Back”), rather than semantic, nature. Since clerical activities constitute only about 20% of the verification effort [GRAH93], the usefulness of those tools is limited. Moreover, the tools usually address isolated aspects of

testing or static analysis, making it practically impossible to use several verification methods simultaneously.

STA methods fall into two major categories. *Static Analysis* analyzes the text of the program, without its execution while *Dynamic Analysis* is execution based. Either method can be either *descriptive* or *proscriptive.* Descriptive techniques are of explanatory nature, offering help in reasoning about the program. The term "reasoning" applies to any kind of semantic analysis, including program walkthroughs and formal verification. In contrast, proscriptive techniques do pass judgment on the code and thus can be used to detect real or potential flaws in it.

Traditionally, static and dynamic analyses have been developing independently of each other. We feel, however, that there are at least two reasons to integrate these techniques. First, static analysis is a prerequisite to dynamic analysis, such as testing and debugging. Clearly, any structural testing technique such as code coverage must be preceded by the identification of the parts of code that need to be exercised during testing, e.g. statements, branches, definition-use chains or context Second, dynamic analysis should be viewed as a refinement of static analysis, helping to resolve the inherent undecidability of the latter, rather than a technique in and of itself..

Our experience with STAD, a System for Testing And Debugging for Pascal programs [LASK90a], suggests that two major problems have to be solved before STA tools achieve a greater acceptance. First, it is the development of a common framework around which otherwise disparate STA methods can be integrated. Second, it is the need for the explanation of the reported findings, although that needs some qualification.

On the one hand, our experience indicates that reports of static analysis will be largely ignored by the users if the reasons for the findings are *too* difficult to grasp. On the other hand, it confirms the findings reported in [HERM76] that the effort to explain reports of static or dynamic analysis significantly increases the understanding of the program and thus can be crucial for its Quality Assessment. For example, many "interesting" *errors are discovered not necessarily through actual testing but during the synthesis of test data*, for a statically identified part of code. One explanation of this phenomenon is that the test synthesis process directs the programmer's attention to some aspects of the program that might have been otherwise overlooked.

The objective of the paper is to study program dependencies as the common conceptual framework for static *and* dynamic STA techniques. The ultimate goal of the research is to derive a family of relations for *arbitrary* Ada programs that not only enjoy the mathematical elegance of the flow relations introduced by Bergeretti and Carre for *structured* programs [BECA85] but also alleviate some inherent and pragmatic limitations of static analysis in general, by providing explanation features and distinguishing between *potential* and *guaranteed* events. Section 3 offers a justification of that goal. Section 4 introduces a modified dependency model for Ada programs that alleviates some of the drawbacks of static analysis discussed in Section 3. In terms of that model, the static Object-Object Potential dependency (OOP) is introduced in Section 5 and its usefulness in the detection of program anomalies and descriptive queries demonstrated. Due to the lack of space, OOP is the only relation described in some detail, together with the method of derivation of its potential and guaranteed versions and the explanation features. In Section 6 an entirely novel path-oriented dependency is proposed and its potential usefulness demonstrated. Finally, further research is discussed in Section 7. It has to be emphasized, however, that the essence of most STA methods is essentially *language-independent* and they simple have to be adopted to handle the idiosyncrasies of a particular language at hand; Ada is no exception.

Throughout, *iff* is a shorthand for “if and only if;”

.

3. STATIC PROGRAM ANALYSIS.

The two best known approaches to static program analysis are program information-flow and program dependencies. The former have been specifically advanced to meet software engineering needs in the salient work by Bergeretti and Carre [BECA85], while the latter have originated in the area of code optimization.

3.1 Information Flow.

Let V and E stand, respectively, for the set of all variables and expressions in the program. For statement S in the program, the following three *information-flow* relations are defined:

ρ, from V to V, where $x \rho y$ *iff* the value of x on entry to S may be used to obtain the value of y on exit from S

λ, from V to E, where $x \lambda e$ *iff* the entry value of x may be used to evaluate expression e in S

μ, from E to V, where $e \mu x$ *iff* the value of expression e in S may be used to obtain the exit value of x.

The qualifier *may be* signifies the fact that the flow is potential, rather than guaranteed, meaning that there is a path through S along which the concept being defined actually “happens.” However, that is not necessarily true for the ρ relation when x=y. Clearly, when $x \rho x$ holds, either x is used to compute x, e.g. x:=x+2, or x is preserved by S, i.e. there is assignment to x in S, or both. This fact is reflected in the following expression for the ρ relation in terms of the remaining two:

$\rho = \lambda\mu$ {(x x)| variable x is preserved by S}.

3.2 Program Dependencies.

The dependencies are defined in terms of the routine *flowgraph* and capture the potential impact (a conservative, statically derived estimate) of one program entity onto another during execution. (The entities in questions are program statements and variables and have no relation to Ada WITH-ed dependencies between compilation units).

Two basic dependencies between program statements have been defined in the literature on code optimization. There is a *Data Dependency* from statement p to statement q iff there exists variable X that is defined in p, used in q and there exists a path from p to q along which X is not redefined. Thus, the action carried out by q depends potentially (i.e. when the path involved is actually taken) on the value of X assigned in p. Statement q is *Control Dependent* on p iff p is a decision statement and there exist two *acyclic* paths from p to q, say w1 and w2, such that (1) q cannot be reached from p on w2 and (2) q *can* be reached from p on w1 and there is no other decision statement on w1. In other words, if q is control dependent on p then the selection of q for execution depends on the run-time evaluation of the predicate in p. For example, in the code: IF p THEN A; ELSE B; END IF; E, if A and B do not contain decision statements (i.e. each is a straight-line code), each statement in A and B is control dependent on p, while E is not. A formal definition of control dependency can be found elsewhere [FEOW87, OTOT84, POCL90]. Mathematically, data and control dependencies are binary relations on the set of nodes in the flowgraph and as such can be represented by directed graphs themselves. In particular, when only one graph with two distinct sets of arcs for each relation is used, one obtains the *Program Dependence Graph (PDG).* The dependencies represent the nearest, one-step concepts. The *transitive closure* of a dependency corresponds to paths in the PDG, i.e. to chains of dependencies, referred here to as the statements' *scope* of data or control. That information can help the programmer in reasoning about the program and also to guide structural testing, debugging and projecting the effects of program modifications.

3.3 Limitations of Static Analysis.

At least three major problems of static analysis have been identified: Its inherent undecidability, its (at the current state of the art) inability to offer an explanation of the reported descriptive or proscriptive events and its inadequacy to handle events that occur on individual program paths.

3.3.1 Undecidability.

A problem is undecidable when there exists no terminating algorithm that solves it. Thus, when static analysis shows that variable X affects variable Y, such a finding is necessarily conservative, meaning rather that "X *may* affect Y." For example, for the statement Y := X-X, static analysis will establish that the variable X affects the variable Y, although there is no semantic dependence between them. Another example is the potential infeasibility of a path along which a data flow anomaly occurs:

```
-- no assignment to A
i:=1; n:=5; WHILE i<=n LOOP get(A(i)); i := i+1; END LOOP;
-- path with no assignment to A
X:=A(5);
```

Static analysis may issue a warning that an undefined value of A may be used in the program. Such an event may be detected by establishing that variable A is *live* at the entry to the segment (i.e. there is a path from the start to the exit on which A is used before being redefined, [HECH77] but A is undefined at the start. Indeed, i:=1; n:=5; X:=A(5) is the offending path. However, the path is infeasible, i.e. it cannot be traversed at all but that fact cannot be established by static analysis.

Yet another case when static analysis runs into undecidable problems is *array handling*. Clearly, an assignment to or use of array element is considered, respectively, an assignment to or use of the entire array. For example, for the following sequence of statements:

```
A(i):=X; A(j):=Y;
```

the second statement will be classified as "killing" the first one. Consequently, the first statement will not be classified as having an impact on the final value of A.

3.3.2 Potential vs. Guaranteed Dependencies.

As originally defined, the dependencies establish the potential *existence* of some events, without guaranteeing that those events will actually happen. Consider, for example, the following statement:

```
IF C THEN Y:=f(A); X:=g(B); ELSE Y:=f(B); X:=g(A); END IF; E;…
```

Static analysis will identify the variables X and Y at E as data dependent on both A and B at the beginning of the statement, despite the fact that, depending on the concrete execution, Y depends only on A and X depends only on B or vice versa. The following statement illustrates how events on concrete path through the code affect the detection of data flow anomalies:

```
<<S>>
```

GET(A,B); IF path(A,B) THEN X:=f(A,B); ELSE Y:=f(A,B); END IF;

<<E>> U:=g(X,Y);

Standard analysis will at best signal a *potential* use of undefined X or Y in U:=g(X,Y). However, there is in fact a *guaranteed data flow anomaly* at E since there is no path from S to E along which *both* X and Y are defined when E is reached.

The above examples show that static analysis fails to identify events on particular program paths. This is a serious drawback since the program's operational semantics and, consequently, testing and debugging, are defined in terms of the computations along individual paths through the program. Path relations, introduced in Section 5, offer a solution to the problem.

3.4 Applications in Software Engineering.

Flow relations offer a solid mathematical foundation for the analysis of structured programs. Indeed, they are used in the SPARK approach, whose Examiner does carry out rather advanced proscriptive analysis and some formal verification for a subset of Ada [BARN97][1]. As SPARK requires user-supplied "annotations" that express the *intended* information flow patterns (to be checked against the actual ones), SPARK is a language in its own rights, best suited for top-down software design. Certainly, flow relations can potentially be used for descriptive analysis and *without* the need for annotations. However, there are several reasons that mitigate against the use of flow relations as *the* unifying framework of an STA system, even if such a system is targeted for programs written in a "real" subset of Ada, cf C-SMART [ROMA97].

First, the language restrictions would severely narrow the class of analyzed programs, since even the most dangerous features will eventually be used. Second, our experience with STAD strongly suggests that in practice reasoning about programs often involves analysis of events on individual program paths, rather than the orthogonal approach of Hoare's assertion-based verification method; that, however, is difficult to support by a pure information flow model. Third, it is not clear how to use that model to support dynamic analysis that is based upon the notion of program trace, a recorded history of actually executed path [KOLA90]. Fourth, the very definition of the ρ relation does not allow one to identify variables preserved by a statement.

Since program dependencies have been originally formulated for code optimization, their use in software analysis is not well established, at least explicitly. Although Podgurski and Clarke [POCL90] propose the use of the dependencies as a common framework for software testing, static analysis, debugging and maintenance, most practically used methods are expressed in terms only indirectly related to the dependencies. For example, the most popular *statement coverage* testing requires that "every statement in the program be exercised at least once." The implicit assumption here is that the output statement (e.g. RETURN) depends on all other statements in the program, a fact not necessarily true.

4. MODIFIED DEPENDENCY MODEL.

In this section we propose several modifications to *static* program dependencies that alleviate some of the deficiencies discussed above. They include different treatment of used and defined array variables, the notions of potential and guaranteed dependencies and some i*nter*procedural concepts. We believe that it is important to separate control flow from data transformation, since that better corresponds to the way programmers analyze their programs. Also, that separation is a must for the dependencies to serve as a blueprint for dynamic analysis. Since concrete dependency models will vary depending on particular applications, Object-Object relation and its derivatives illustrate the approach. We start with a few comments on the explanation feature followed by the derivation of Ada flow graphs as the basis for dependency analysis of Ada programs.

4.1 Explanation Feature.

The actual form of the justification depends on the problem at hand. For example, if it is reported that the variable X in the assignment Y:=X may be undefined when the statement is reached, it is essential for the user to identify an offending path in the program along which the statement is reached without an intervening assignment to X. Such a path will be referred to as a *trigger path*, since its execution will ultimately lead to the program's failure. Moreover, to facilitate defense programming, the user might be interested in all statements in the program that lie on some trigger paths. Such a set of nodes will be referred to as the *trigger set* of statements: If a trigger statement is traversed on an execution, the event in question may happen. Consequently, the programmer may provide for exceptions to be raised.

Ideally, traversal condition for a trigger path should be derived, i.e. a predicate on the program state which, if true, guarantees that the path will ultimately be taken. However, the derivation of path traversal conditions is only possible if the statements on the path are assignments, and the method of backward substitution can be applied [BACK86]; the occurrence of procedure calls makes that technique impractical. Two approximations to path traversal condition are possible, however. First, it is an ordered tuple of Boolean values of the test statements on the path, which are satisfied if the path is traversed. This

[1] This fact illustrates the importance of language-independent research.

can always be done if the arcs in the flow graph for the program are suitably labeled with logical values. Second, it is the set of variables at the beginning of the path whose values determine the actual traversal. That is easily derivable if the nodes of the flow graph are labeled with the sets of used and defined variables.

4.2 Flow Graphs for Ada Routines.

The flow graph of a routine is a quadruple (N,A,S,E) where (1) N is a set of *vertices* (nodes) that correspond to single entry single exit segments of code at a certain level of decomposition, usually "instructions," i.e. the smallest, executable parts of program's statements, (2) A is a set of arcs (*edges*), a binary relation on V, an edge (n m) denoting a potential transfer of control from n to m, and (3) S and E are, respectively, the start and exit node of the flow graph.

Nodes and labels can be labeled with information needed for the analysis. For dependency analysis, nodes are labeled with two sets of variables - the variables defined and used in the nodes, tagged either partial or total. Arcs of decision statements are labeled with the Boolean expressions which, if true, guarantee their traversal; that approach, rather than the use of Boolean values *true* or *false*, allows a uniform treatment of CASE statements.

Most nodes in the flow graph correspond to single statements (e.g. x:=y) or expressions (e.g. for IF path(x) THEN .., the expression path(x) corresponds to a single predicate node) in the program. However, to guarantee that the flow graph correctly represents real events in the program, extra system nodes have to be created. Here are some examples. For the short circuit control form IF p AND THEN q THEN X;Y; there are four nodes p, q, X and Y with arcs (p Y), (p q), (q X), (q Y) and (X Y). Nodes have to be created to properly capture the default values of routines' formal parameters (if any) and at the same time allow those values to be potentially overwritten by a call to the subprogram. Nodes also have to be created for any local variable declaration with initialization. In the case of a FOR loop and a DECLARE block, the loop control variable and variables declared in the block have to be marked as created on entry to and destroyed on exit from the loop or block. Package initialization routines, prefixed by variables' declarations and initializations, if any, have to be put together in an elaboration order and a flow graph derived for the entire sequence. For the interprocedural analysis, that package elaboration flow graph is the first one to be implicitly invoked before any other subprogram is called.

4.3 Used and Defined Variables.

The derivation of the sets of Defined and Used variables for nodes in the flow graph poses some difficulties, particularly for arrays and procedure calls. Ideally, one would like to treat array entries as independent variables, as is the case in dynamic analysis. However, due to the undecidability of determining the values of array indices, this is impossible in general. What is sometimes possible is the detection of whether only part of the array is defined. That allows one to identify potential non-killing definitions and signal uses of potentially undefined arrays. Thus, if A is an array, an assignment A(i):= <exp> will be referred to as *partial*, capturing the fact that only one entry of A has been defined. For a whole array assignment A:=B, where B is another array, the definition is considered *total*. This is notwithstanding the fact that A and B may be of different length, since that fact cannot be usually statically established. All that can be done is to signal such a possibility and use dynamic analysis (testing) attempting to raise an exception. The case A:= (aggregate) is similar for it is not always possible to determine statically whether the length of the aggregate expression is equal to that of A (observe that in SPARK aggregates have to be qualified and, therefore, their boundaries are statically known [BARN97]). However, if the OTHERS is used in the expression, the definition *cannot* be partial - at worst, the aggregate can be longer than the array. Consequently, in such a case the definition is considered total. In the absence of OTHERS, the definition *may be* partial and the analyzer can issue a suitable warning. For the dependency analysis, however, it may be classified as total anyway, since the analysis is based upon the assumption that every path through the program is potentially feasible, i.e. can be executed for some input data.

The identification of variables used and defined in a subprogram call is also undecidable, due to the fact that different variables may be manipulated on different paths through the subprogram. Therefore, the following approximation is adopted. A variable is used in the call if it corresponds to an IN OUT parameter or appears in the expression that corresponds to an IN parameter. Variables corresponding to OUT parameters are considered defined by the call. In Section 5 we show how path analysis can offer a more refined approach to procedure calls. Again, partial and total definitions apply to array parameters. For the *body* of a subprogram (as opposed to a call) all IN and IN OUT formal parameters are assumed defined, as they are in a *correct* program. Indeed, it is one goal of the *inter*procedural analysis to establish whether some actuals can be undefined in a call to the subprogram.

5. OBJECT-OBJECT POTENTIAL DEPENDENCY OOP.

This is a fundamental, first level or "nearest" data dependency. Let O be a set of objects in the program, i.e. the union of the set V of variables and set C of constants, cf [BARN95]. One can visualize the set C of constants as being (permanently) defined at the entry to the program and available throughout the program. Informally, for nodes p and q in the flow graph, object X and variable Y, the value of Y at (top of, entry to) node q depends on the value of X at (the top of) p, symbolically (Y q) OOP (X p), iff the

value of X at p *may be* used to define the value of Y at q in exactly one place between p and q. Or, alternatively, "there is a path from p to q on which there is an assignment k: Y:=f(X), such that X is not redefined between p and k (always satisfied if X is a constant), and the value of Y assigned in k is available at q. Observe that it is not necessary for X to be defined in p nor used in q.

Here are some descriptive queries supported by OOP. "If the value of X at p is changed, or a statement that modifies X is inserted in front of p, what is the *immediate* scope of propagation of the modification?" An answer to this "forward " query is a set of pairs (Y q) such that (Y q) OOP (X p). Conversely, for a backward query "identify all statements and objects that *directly* affect the value of Y at q," the set of pairs (X p) such that (Y q) OOP (X p) is the answer.

The OOP relation can also be used to formulate and test program *Fault Hypothese*s, i.e. conjectures about potential sources of program incorrectness. For example, if Y at q is found incorrect, one can hypothesize that the code between p and q is correct and then identify all possible immediate objects X whose *incorrect* values at p may cause incorrectness of Y. Alternatively, if the state at p *is* found correct then one can identify the *last* assignments to Y which are potentially faulty, under the assumption that the control flow from p to q is correct.

5.1 The Closure(OOP).

The OOP relation represents "one-step," immediate dependencies between program objects. In practice one is also interested in higher level dependencies, asking, for example, the question, "Which input variables are potentially used in the computation of variable Y at the exit?" An answer to that question is provided by the transitive closure of OOP defined as follows.

Let OOP^1 stand for the first power of OOP, i.e. OOP itself. Then the "second-power" of OOP is defined as follow

(Y q) OOP^2 (X p) iff there exist node r and variable Z such that (Y q) OOP^1 (Z r) and (Z r) OOP^1 (X p). Clearly, there is a second-level dependence of Y at q on X at p because Y at q depends (first level) on Z at r which, in turn, depends (first level) on X at q.

By considering the union of all powers of OOP (there is always a maximum power above which no higher new dependencies exist) one gets the *transitive closure* of OOP, which denotes "chain of dependencies of arbitrary length." Thus

(Y q) Closure(OOP) (X p) iff exists k, k>0, such that (Y q) OOP^k (X p).

The Closure(OOP) relation is similar to the information-flow relation ρ discussed in Section 3. In contrast to the latter, however, Closure(OOP) can be derived for arbitrary control structures, includes constants in the domain of the relation and does not include variables that are not redefined (preserved) between p and q. Thus, if (X q) Closure(OOP) (X p) then the value of X at p is certainly used on a path from p to q to obtain the value of X at q. In contrast, if there is no Z such that (X q) Closure(OOP) (Z p) holds, X is certainly preserved on all paths from p to q.

Observe that if X or Y is an array, the issue of partial vs. total definition comes into play. Clearly, if X is a partially defined array then Y may be defined using an undefined portion of X, a potential anomaly. If X is defined totally, and Y is also an array and the use of X is partial, then Y is also defined partially, although there is no anomaly. The relations should be tagged indicating whether partial, total or *both* types of dependencies apply. Clearly, if (Y q) OOP (X p) holds, Y can depend on X either partially or totally, depending on the path from p to q. Observe that when the OOP relation is represented by a directed graph then, for nodes (X p) and (Y q) in the graph, there can exist two arcs between these nodes that correspond to the two types of dependency.

The Explanation Feature for OOP will be explained below, as it is a byproduct of its derivation.

5.2 Derivation of OOP.

The definition of OOP leads to a simple method for its derivation. Observe that OOP (V _ N) _(V _N) and (Y q) OOP (X p) iff there exists node r such that Y is defined in r using the value of X at p and that value of Y is available at q without a change. In other words, OOP is the composition of two relations: Reaching Definitions RD and Live Uses LU.

To define reaching definitions, one needs an intermediate concept of *killing definition*.

5.2.1 Killing and Nonkilling Definitions.

For a scalar variable X, an assignment X:=… kills any definition of X that reaches the assignment. Thus, no other definition is available at the bottom of the assignment, after it completes execution.

A partial definition of array A, k:A(i):=... , does not kill any definition of A that reaches k; it simply "updates" it. That convention mirrors the fact that only the ith entry of A is changed, while the remaining entries may have values coming from other places.

A total definition k:A:=<exp> *kills* all other definitions that reach k.

Observe that if the exp above is an array B, many definitions of B can reach the assignment, being either partial or total. Thus, *B itself may be defined only partially* when the statement is reached. However, that fact cannot be established at the graph building stage and thus has to become one of the goals of static analysis. When the analysis fails to deliver a clear-cut decision, it might be useful to carry the analysis twice, *assuming* both cases - total and partial definition.

5.2.2 Reaching Definitions Relation RD.

The RD relation is defined as follows: Given variable Y and nodes r and q, (Y q) RD r iff Y is defined in r and, i.e. there is a path from top of (entry to) r to top of q along which Y is not *killed* (overwritten). In other words, the definition of Y in r *reaches* q, another way of stating that the value of Y at q *may* come from r, depending on the path taken from r to q. Observe that it is irrelevant whether Y is manipulated in q or not. When that is the case, however, one obtains a *definition-use chain* for X, cf [HECH77].

Thus, RD (V_N) _N, i.e. RD is a relation from V_N to N; it can be derived by means of a simple bit-vector, iterative top-down algorithm [HECH77].

5.2.3. Live Uses Relation LU.

A counterpart to the RD relation is the *Live Uses Relation LU*, LU N _(V _N), defined as follows: r LU (X p) iff variable X is used in r and there is a path from top of (entry to) p to top of r along which X is not redefined. In other words, X at p might be used in r without being changed between p and r. Although the notion of the LU relation appears new, its derivation is essentially a bottom-up version of the RD algorithm. The only difference is that no use is considered "killed."

Now, the OOP relation can be defined as follows:

(Y q) OOP (X p) iff exist node r such that
Y is defined in r,
(Y q) RD r, -- value of Y defined in r reaches q, and
r LU (X p) -- X at p is used in r without being modified

Thus, OOP can be derived by first deriving RD and LU and taking their composition, i.e. OOP = RD; LU. Observe that the intermediate node r above is not necessarily unique.

5.3 Explanation Feature.

In terms of the above equation, the explanation feature for the RD relation is a *shortest* trigger path from r to q along which Y is not redefined, or all shortest paths of the same length. Observe that if the node r: Y:=… above lies on a cycle (within a loop) the number of paths along which the definition of Y in r reaches q may be infinite. In such a case, the identification of *all* paths along which Y is not redefined is not feasible without the use of a compressing technique such as, for instance, the language of regular expressions. However, our experience with that approach is not encouraging [LASK90b]. The trigger set of RD is the set of nodes on any path from r to q that reach q without encountering a definition of Y. For the LU relation, a trigger path is one from p to r along which X is not redefined. Since OOP=RD; LU, the explanation feature for OOP can be composed of the features for RD and LU. In principle, the intermediate nodes r above do not have to appear in the explanation of OOP. In practice, however, it may be very useful to identify such nodes.

A method for the derivation of the explanation is similar to the one in Section 5.5 below for the detection of data flow anomalies.

Although the RD and LU relations are used to derive the OOP relation, they also support specific queries about the program. Here are some examples. "Where the last definitions of X that reach q may come from (debugging)?" "Where a definition of X in p can be directly used (forward propagation of action in p)." Also, the following potentially useful sets of variables are easily derived:

PotentiallyDefined(q), i.e. all variables *potentially* defined at q. Variable v is in PotentiallyDefined(q) iff exists node d such that (v q) RD d.

Undefined(q), i.e. variables *always* undefined at q; Undefined(q) = V-PotentiallyDefined(q).

Live(q) (live variables at p), i.e. those that can be used on some path that originates at q before being redefined, if at all (formulating error hypotheses in debugging: only a live variable at q, if incorrect, may cause the program's failure). Variable v is in Live(q) iff exists node u such that u LU (v q).

5.4 Derivative Dependencies.

Several other dependency relations can be derived from OOP. For example, if there is no X for which (Y q) OOP (X p) holds then Y is never modified (is preserved) between p and q. This gives rise to the *guaranteed* preserved relation PRG defined as (Y q) PRG (Y p) iff for every X, (Y q) OOP (X p) does not hold. A stronger version of OOP is OOG, a *guaranteed* data dependence, defined as (Y q) OOG (X p) iff X is used to define Y once on *every* path from p to q. Now, *potential* preserved relation PRP is defined as follows: (Y q) PRP (Y p) iff there exists variable X such that (Y q) OOP (X p) holds and (Y q) PRG (Y p) does not. That is, if (Y q) PRP (Y p) holds, Y is preserved only on some paths from p to q. The identification of preserved variables is important in program debugging when a fault hypothesis is formulated and tested. For example, knowing that (1) Y at q is incorrect, (2) Y at p is correct and (3) Y is always preserved between p and q, one concludes that there is a missing assignment to Y between p and q.

Assuming that the OOP and OOG relations are available, the above definitions of PRP and PRG offer a method for their derivation. The derivation of OOP has been discussed in Section 5.2. The OOG relation can be derived by suitable modification of a standard path analysis algorithm, such as Dijkstra's or Floyd's shortest path or Warshall's transitive closure. Such a modification would aim towards establishing the existence of a path from p to q that does not traverse any assignment of the form Y:=f(X).

Similarly to OOP, higher powers and the closure of these relations can be defined and computed using the Warshall's algorithm.

5.5 Detection of Program Anomalies.

The detection of program anomalies is perhaps the most typical application of *proscriptive* static analysis. A *Data Flow Anomaly* is a sequence of actions on a variable in the program that is either erroneous or suspected to be so. Typically, the following sequences of actions are considered anomalies, since they are either incorrect or symptomatic of error:

UR (Undefined-Referenced) anomaly: A variable is not defined but used, e.g. y:=f(x) with no intervening assignment to x**.** *DD (Defined-Defined)* anomaly: A variable is defined and redefined again, without being used in between, e.g. x:=4 followed by x := 3 with no intervening use of x in between. *RD (Redundant Definition)* anomaly: A variable is defined but not used afterwards, e.g. x := x+3 is not followed by a use of x. An anomaly is *feasible* if the offending path can be executed for some input data. It is obvious that if UR anomaly is feasible then there is an error in the program. In contrast, the DD and RD anomalies, whether potential or guaranteed, are only symptomatic of an error. For example, they can be due to some code left unintentionally during program modifications.

To alleviate some of the limitations of static analysis discussed in Section 2, the above concepts are refined as follows. A data flow anomaly is *potential* if it occurs only on some paths in the program. Therefore, if the offending path is infeasible (cannot be executed), no real error in the program occurs. If, however, a data flow anomaly occurs on every path in the program, then the anomaly is *guaranteed.* Moreover, either kind of anomaly can be *partial* if a partially defined array or record is involved. However, to simplify the analysis partial anomalies will not be discussed in this paper.

It is also important to identify (1) the set of *trigger nodes* from which the anomaly in question can be potentially activated and (2) at least one *trigger path* along which such an event can happen. The approach will be illustrated for the UR anomaly only. In what follows, path(p,k,n,q) stands for a path from p to q that traverses nodes k and n, in that order.

5.5.1 Guaranteed UR Anomaly.

This anomaly occurs at node q (symbolically URG(q)) iff there exists variable v that is used in q and is undefined at q on any path(S,q) from start to q. In terms of the OOP relation the URG anomaly occurs when Y is used in q and there is no pair (X p) for which (Y q) OOP (X p) holds. (Observe that if constant objects were not included in the range of OOP, spurious anomalies would be reported).

The set of trigger nodes Tn-URG(q) is simply a set of nodes from which n can be reached, i.e. which lie on some path(S,q). A trigger path is any path(S,k,q), where k is in Tn-URG(q).

The detection of the above is straightforward. Clearly, URG(q) occurs iff there exists variable v, such that v is used in q and is in Undefined(q), i.e. no definition of v reaches q. Node k is a trigger node for URG(q) if k lies on some path(S,q). Both Tn-URG(q) and a trigger path(S,k,q) are easily identifiable by a standard path analysis algorithm.

5.5.2 Potential UR Anomaly.

This anomaly occurs at node q (symbolically URP(p)) iff there exist variable v and path(S,q) such that (1) v is used in q, (2) v is undefined on path(S,q) and (3) v **is** defined at q, i.e. at least one definition of v reaches q. Due to condition (2), the detection of URP(q) is more contrived than that of URG(q). The following simple modification of a path analysis algorithm offers a solution to the problem: Identify the existence of path(S,q) that does *not* traverse a definition of v. If such a path does exist, and

some definition of v reaches q, there is an URP anomaly at q. The proposed modification can also be used to identify the set Tn-URPATH(q) of trigger nodes and identify a shortest trigger path(S,k,q) such that k is in Tn-URP(q).

5.6 Interprocedural Dependencies.

As defined above, the dependencies are of *intra*procedural nature, i.e. apply to single subprograms, including global objects. To be useful in programming practice, however, they need to be extended onto a collection of procedures. Here is an example of such an extension of the OOP relation, denoted SOOP (S for System):.

Variable Y at node q in procedure B depends on object X at node p in procedure A, symbolically (Y q B) SOOP (X p A) iff (1) there is a call A(Y,…) in B, in which Y corresponds to an IN OUT or OUT parameter of A, (2) X in A is used to compute a definition of the formal parameter in A that corresponds to Y in the call and (3) the definition of Y in the call reaches q. Again, the relation should be split into potential vs. guaranteed and their closure computed. The *(Subprogram Call) Graph* , in which every procedure is a node and there is an arc from A to B iff A may call B [HECH77], together with the flow graphs for the subprograms involved, are natural vehicles for the computation of the SOOP.

6. PATH RELATIONS.

Path relations identify program variables that are manipulated on individual paths through the program. In this section path relations are defined and some of their potential applications discussed; however, no formal derivation algorithm is presented. The approach will be illustrated by the program in Figure 1.

```
1: S: get(A,B,X,Y);
2:   if c(A,B)
     3: then X:=e1(A,B); A:=e2(Y);
     4: else Y:=e3(A,B); B:=e4(X); end if;
5:   if p(A,B)
     6: then X := f(A); U := g(Y); T := r(A,Y);
     7: else Y := h(B); W := z(X); Q := s(B,X); end if;
8: E: Z : Put(X,Y).
```

Figure 1. An example program.

6.1 Motivations.

To get the essential intuition, consider first the following ***block*** B of a straight-line sequence of statements that are always executed together:

B: x:=f(a,b); y:=g(c,x); z:=h(a,d).

In static analysis variables live in the block are treated as used on entry to the block while all variables defined in a statement in the block are considered defined in the block [HECH77]. Thus, USED(B) = {a,b,c,d}, DEFINED(B) = {x,y,z}. However, that model suggests spurious dependencies between variables in those sets since, in fact, there is only "partial" dependence between them. That is, x depends on (a,b), y depends on (a,b,c) and z depends on (a,d). It is precisely the identification of those partial dependencies that underlies the notion of path relations.

6.2 Definition of Path Relations.

Let V be the set of variables in the program and p be a path through statement S in the program. ***Path relation*** **PR** *for path p* is a binary relation on V, i.e., PR is s subset of V_V, such that xPRy iff x at the beginning of p is used to compute the value of y that reaches the end of p. According to our principle of separating control flow from data transformation, PR will be structured as the union of (not necessarily disjoint) ***data path relation*** **DR** and ***control path relation*** **CR**. Consider for example path p=(1,2,3,5,7,8) through the program in Figure 1. Since the path is the concatenation of two paths p1=(1,2,3,5) and p2=(5,7,8), path relations can be derived first for p1 and p2 and then composed to yield the relation for p. Considering first DR, there is DR(p1)={(A X), (B X), (Y A)} and DR(p2)={(B Y), (X W), (X Q), (B Q)}. Then PR(p) is obtained by composing DR(p1) and DR(p2). This is done first by identifying those pairs in DR(p1) whose second element is the same as the first element of a pair in DR(p2), if any, e.g. (A X) and (X W) and then replacing them by (A W). Pairs that cannot be replaced that way simply remain part of the composition. Thus, DR(p)={(A X), (A W), (A Q), (B X), (B W), (B Q), (Y A), (B Y)}. CR(p1) illustrates the meaning of the control relation: CR(p1)={(A X), (A A), (A Y), (A B), (B X), (B A), (B Y), (B B)}. Informally, pair (X Y) of variables is in CR if Y can be defined or not depending on a control decision that uses X.

6.3 Functional Representation.

Before demonstrating the usefulness of path relations, they will be represented in a more compact functional form, as a set of data *maplets*. Let S be a programming statement. A ***(data) maplet*** m is an ordered pair m=<A B>, where A and B are subsets of variables from V, such that *every* variable in A on entry to S is used to compute *all* variables in B on exit from S. A and B are, respectively, the *input* and *output* set of the maplet.

Thus, the relation DR(p) above, for p=(1,2,3,5,7,8), has the following functional representation (input and output maplet sets are enclosed within the parentheses (), while particular relations, sets of maplets, are within square braces [ ]; braces {} are reserved for sets of path relations, e.g., for program statements):

DR(p)= [< (A,B) (X,Q,W)>, < (Y) (A)>, < (B) (Y)>]

Given statement S in the program one has to consider every path through S. This leads to a set {PR1,...,PRn} of path relations associated with S. Each PRi in that set corresponds in general to a nonempty *class of paths* through S; clearly, several paths may have the same path relation, if only they exhibit identical patterns of used and defined variables. When there is potentially infinite number of paths through S (in the case of loops), the number of path relations is always finite due to a finite number of variables in the program. The basic technique for the derivation of path relations is the composition of relations derived for constituent statements. Figure 2 shows some path relations for the program in Figure 1. Observe that some statements in the program are built from other statements. For example, statement 2 (the first **if**) is built from statements 3 and 4 and the predicate in 2. Consequently, the DR_2 relation is the union of DR_3 and DR_4.

DR_3 = {[<(A,B) (X)>, <(Y) (A)>]}, CR_3 = {}

DR_4 = {[<(A,B) (Y)>, <(X) (B)>]}, CR_4 = {}

DR_2 = {[<(A,B) (X)>, <(Y) (A)>], [<(A B) (Y)>, <(X) (B)>]}

CR_2 = {[<(A,B.) (X,A,Y,B)>]}

DR_6 = {[<(A) (X)>, <(Y) (U)>, <(A,Y) (T)>]}, CR_6 = {}

DR_7 = {[<(B) (Y)>, <(X) (W)>, <(B,X) (Q)>]}, CR_7 = {}

DR_5 ={[<(A) (X)>, <(Y) (U)>, <(A,Y) (T)>], [<(B) (Y)>, <(X) (W)>, <(B,X) (Q)>]}

CR_5 = {[<(A,B) (X,Y,T,U,W,Q)>]}

Figure 2. Path relations for some statements in Figure 1.

6.4 Applications of Path Relations.

In this subsection we show how path relations can be used to provide a more accurate model of procedure calls, detect data flow anomalies not tractable by standard methods and help formulate and test fault hypotheses.

6.4.1 Procedure Calls.

If X is an actual argument (parameter) of a call to procedure P, it is in general undecidable whether X is used or defined in P. The use of IN and OUT qualifiers in Ada solves the problem only partially by enabling the compiler to check whether there is at least one use of an IN variable and at least one definition of an OUT variable. Thus, in data flow analysis a crude approximation is usually adopted, e.g. variables passed by reference are assumed dependent on those passed by value. Such an approximation may leads to spurious dependencies, e.g. an OUT variable may depend only on *some* IN parameters on *some* paths though the procedure. Path relations for the procedure's body provides a more refined model of the data exchange patterns through the procedure and might alleviate some undecidable problems of interprocedural analysis. That approach might also open a way for handling object oriented programming paradigms.

6.4.2 Detection of Anomalies.

To illustrate the detection of anomalies, consider the following code:

```
GET(A,B);
IF path(A,B.) THEN
   X:=f(A,B);
ELSE
   Y:=f(A,B);
END IF;
U:=g(X,Y);
```

The set of data path relations for S1 is RO={[<(A,B) (X)>], [<(A,B) (Y)>]}, indicating that there is no path

along which *both* X and Y are defined. Thus, a ***guaranteed data flow anomaly*** at the last statement has been detected. In contrast, "classical" flow analysis would at best identify a *potential* data flow anomaly, due to the possibility of an undefined X or Y being used in the last assignment.

6.4.3 Error Propagation Hypotheses.

Path relations offer a possibility to analyze error creation and propagation along program's paths. Assume that the variables X and Y at the exit E of the program in Figure 1 are incorrect; their incorrectness can be either an established fact (debugging) or postulated (testing). In debugging one wants to identify those variables at 5 which *explain* the error at E, i.e. whose incorrect values at 5 may propagate to X and Y at the exit E. In testing one wants to identify those variables in order to synthesize tests that cause the program to fail, with X and Y incorrect (the main goal of testing is to demonstrate program's incorrectness).

Let FH_2 be the Fault Hypothesis stating that the first **if** statement at 2 is faulty; this is equivalent to saying that a *primary* error has occurred at the exit of the statement and has propagated through the *correct* second **if** to the exit E. Path relations DR5 and CR5 for the latter (Figure 3) are used to formulate a ***Primary Error Hypothesis (PEH)*** at node 2 and the corresponding ***Error Propagation Hypotheses (EPH)*** through the second **if**. Clearly, if there exists a path relation in DR5 or CR5 with a maplet <I O> such that {X,Y} is a subset of O then X and Y can be rendered incorrect at E on a *single execution* of the second **if** (EPH), if a variable in I is affected by a primary error (PEH). However, if there is no such path relation but there exist two path relations in DR5 or CR5 such that only X or Y is in O, then either two executions are needed to render X and Y both incorrect or one of them is already affected by a primary error at the entry to statement 6. Thus, one gets the following PEHs and associated with them relevant EPHs:

PEH1: Either A or B at E_C is incorrect (CR5).

*EPH1.1: A decision (secondary) error in path(A.B.) occurs, causing X, Y, U, T, W, and Q incorrect on a single execution, regardless of the branch taken through the second **if**.*

PEH2: Y and A at E_C are incorrect (DR5).

*EPH2.1: No control error in the second **if**, the **then** branch entered correctly. Incorrect A at E_C, causes a secondary error in X and T, while Y is incorrect due to a primary error at E_C, persists through the branch (no redefinition of Y) and propagates to U.*

PEH3: X and B at E_C are incorrect (DR5).

EPH3.1: Similar to EPH2.1, with A, X, Y, T,U replaced by B, Y, X,Q,W in that order.

Some of these PEHs can be rejected by the programmer if the whole scope of primary error propagation is taken into account. For example, if variables U, T, W and Q are all correct at E, PEH1 can be rejected; if U and T are correct then PEH2 can be rejected. This is based on the assumption that it is unlikely that errors in all potentially affected variables are masked.

7. CONCLUSIONS.

The working hypothesis of this paper has been the belief that Program Dependencies provide the best common framework for the integration of various Software Testing and Analysis (STA) methods. They offer the same degree of static semantic support that does the information-flow model proposed by Bergeretti and Carre [BECA85] but also, due to the clear separation of control flow from data flow and the lack of language restrictions, they are potentially applicable to a wider class of STA methods, including dynamic (execution-based) analysis. The latter's importance cannot be overestimated, since dynamic analysis is sometimes the only way to resolve the inherent undecidability of static analysis. However, since program dependencies have originally been formulated for compiler optimization, their uncritical use in software engineering is lacking. Therefore, modifications to the original dependencies have been proposed in this paper. They include partial vs. total definitions of Ada arrays, new concept of reaching definitions, potential vs. guaranteed dependencies, *inter*procedural dependencies, and an explanation feature that helps the user understand the reasons for the generated reports. It has been shown how the modified model can support descriptive and proscriptive (e.g. anomalies) queries about the program. Also, Path Analysis, a novel method for the identification of dependencies along individual program paths has been proposed. It has been shown that path analysis offers a more accurate model for procedure calls, allows one to detect otherwise undetectable data flow anomalies and can serve as a vehicle for the analysis of error creation and propagation in testing and debugging

The modified dependency model has been illustrated by the OOP (Object-Object Potential) dependency, which roughly corresponds to the variable-to-variable flow relation ρ [BECA85]. This is, however, only the first step and other dependencies have to be developed. Besides their counterparts of the λ and μ flow relations [BECA85] they should address the issue of combining data dependencies with control dependencies allowing, for example, an answer to the following query: "Which variables at p affect the selection for execution of statement q?"

Moreover, dynamic, execution-based counterparts of those relations have to be defined to form a basis for the integration of static and dynamic methods.

Also, research into system level analysis is needed to provide answers to specific versions of the following general question: "What kind of semantic information about the system can be gleaned by static and dynamic i*nter*procedural analysis?" Thus, one may be interested in the identification of potentially dangerous sequences of calls to procedures in a package, in the identification of high-level data and control anomalies or in the potential propagation of exceptions.

However. the key to the success of the STA methods is the continuing theoretical and experimental research, if one shares our belief that "*There is nothing more practical than a good theory"* (attributed to Boltzmann). Towards that goal, what is needed most is the development of a sound theory of program *in*correctness, rather than correctness. That is, the nature of programming faults, the origination of primary errors, the mechanism of error masking and propagation should be understood first to answer the question, "What does the method do?" The proposed dependency extensions are but a modest step towards such a theory for the procedural level. It is likely that such a theory will be arrived at in a piecemeal fashion, offering solutions to some subproblems before their conceptual integration takes place. Essential here is experimental testing and validation of theories.

8. REFERENCES

[BACK86] R.C.Backhouse, "Program Construction and Verification," Prentice/Hall, 1986

[BARN95] Barnes, J., "Programming in Ada 95," Addison-Wesley, 1995.

[BARN97] Barnes, J., "High Integrity Ada, The SPARK Approach," Addison-Wesley, 1997

[BAHO93] T.Ball, S.Horwitz, "Slicing Programs with Arbitrary Control-flow," in Automated and Algorithmic Debugging, First International Workshop, AADEBUG'93, Linkoping, Sweden, Springer -Verlag, Lecture Notes in Computer Science 749, p.206-222.

[BECA85] Bergerreti,J.F., B.A.Carre, "Information-Flow and Data-Flow Analysis of **while**-Programs," *ACM Trans. on Programming Languages and Systems*, Vol.7, No.1, Jan 1985, pp. 37-61.

[FEOW87] Ferrante, J., K.J.Ottenstein, J.D.Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Trans. Programming Language and Systems," Vol.9*, No.3, July 1987, p.319-349.

[GRAH93], Graham, D.R., " Where is CAST Heading? Directions and Trends for Testing Tools," Proc. Sixth International Software Quality Week, San Francisco, May 25-28, 1993

[HECH77] Hecht, M.S., *"Flow Analysis of Computer Programs,"* North-Holland 1977.

[HERM76] Herman, P.M.," A Data Flow Analysis Approach to Program Testing," *The Australian Computer Journal*, v.8, no.3, Nov 1976, 347-354.

[JONE90] Jones, C.B., *"Systematic Software Development using VDM,"* Prentice-Hall, 1990.

[KOLA90] Korel, B., J.Laski, "Dynamic Slicing of Computer Programs," *Journal for Systems and Software*, 1990, v.13, p.187-195.

[LASK90a] Laski, J., "Data flow testing in STAD," *The Journal of Systems Software*,1990, Vol.12, pp. 3-14.

[LASK90b] Laski, J., "Path Expressions in Data Flow Testing," Proc. Compsac'90, 14th Annual International Computer Software&Applications Conference, Chicago, Il., October 29-November 2, 1990, p.570-576.

[OTOT84] Ottenstein, K.L., L.M.Ottenstein, "The Program Dependence Graph in a Software Development

Environment," *Proc. ACM SIGPLAN/SIGSOFT Symposium on Practical Programming Environments*, Pittsburgh, PA, April 23-25, 1984; ACM SIGPLAN Notices, 19, 5 (May 1984), p. 177-184

[POCL90] Podgurski, A., L.A. Clarke, "A Formal Model of Program Dependencies and Its Implications for Software Testing, Debugging, and Maintenance," *IEEE Trans. Softw. Engineering*, Vol.16 , No. 9, Sep 1990, pp. 965-979.

[ROMA97] Romanski, G. “Safety Critical Software Handbook,” Aonix 1997, ADOC-SCHB-70519