Generating Testing and Analysis Tools with Aria

PREMKUMAR T. DEVANBU and DAVID S. ROSENBLUM
AT&T Research
and
ALEXANDER L. WOLF
University of Colorado

Many software testing and analysis tools manipulate graph representations of programs, such as abstract syntax trees or abstract semantics graphs. Handcrafting such tools in conventional programming languages can be difficult, error prone, and time consuming. Our approach is to use application generators targeted for the domain of graph-representation-based testing and analysis tools. Moreover, we generate the generators themselves, so that the development of tools based on different languages and/or representations can also be supported better. In this article we report on our experiences in developing and using a system called Aria that generates testing and analysis tools based on an abstract semantics graph representation for C and C++ called Reprise. Aria itself was generated by the Genoa system. We demonstrate the utility of Aria and, thereby, the power of our approach, by showing Aria's use in the development of a number of useful testing and analysis tools.

Categories and Subject Descriptors: D.1.m [**Programming Techniques**]: Miscellaneous—*graph traversal*; D.2.2 [**Software Engineering**]: Tools and Techniques—*software libraries*; *tool generators*; *tool specification*; D.2.5 [**Software Engineering**]: Testing and Debugging—*coverage analyzers*; D.2.8 [**Software Engineering**]: Metrics—*complexity measures*; D.2.m [**Software Engineering**]: Miscellaneous—*software analysis*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*control structures*; *data types and structures*; D.3.4 [**Programming Languages**]: Processors—*code generation*; *parsing*; *unparsing*; E.1 [**Data**]: Data Structures—*graph representations of programs*

General Terms: Algorithms, Design, Languages, Verification

Additional Key Words and Phrases: Application generators, Aria, Genoa, program dependence graphs, program representations, Reprise, software analysis, software testing, tools

The work of A.L. Wolf was supported in part by the Air Force Materiel Command, Rome Laboratory, and the Advanced Research Projects Agency under contract number F30602-94-C-0253. The content of the information does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred.
Authors' addresses: P.T. Devanbu and D.S. Rosenblum, AT&T Research, Murray Hill, NJ 07974; email: {prem; dsr}@research.att.com; A.L. Wolf, Software Engineering Research Laboratory, University of Colorado, Boulder, CO 80309; email: alw@cs.colorado.edu.

1. INTRODUCTION

Many software testing and analysis tools are based on specialized manipulations of a graph representation of programs, such as a parse tree, an abstract syntax tree (AST), or an abstract semantics graph (ASG).[1] Such tools traverse and/or modify the graph representations to generate cross-reference information, perform symbolic execution and evaluation, create test data, and instrument code. Typically, they also derive more specialized structures, such as control-flow graphs, from the basic graph representations.

While builders of compilers, interpreters, editors, and the like have numerous construction aids available to them (e.g., Lex/Yacc, Gandalf [Habermann and Notkin 1986], the Cornell Synthesizer Generator [Reps and Teitelbaum 1984], Centaur [Borras et al. 1988], Refine [Reasoning Systems 1990], Pan [Ballance et al. 1992], and Eli [Gray et al. 1992]), the same cannot be said for builders of testing and analysis tools. Typically, testing and analysis tools are individually handcrafted in a conventional programming language such as Ada, C, or Lisp and can require thousands of lines of code. For tools that embody complex algorithms, such as analyses of definition/use relationships, control-flow dependencies, or pointer aliasing, this method can be difficult, error prone, and time consuming. Using high-level specification languages to automate the construction of these tools would clearly be advantageous.

This article describes a system called Aria for generating ASG-based testing and analysis tools for C and C++. Aria is essentially an application generator that can create powerful tools from terse, high-level specifications of the graph manipulations that those tools are intended to perform. For instance, the following small Aria specification describes a tool that lists all assignments to, and uses of, variables appearing in a C or C++ program.

```
[(? NameRef
  (IF (AND (TYPEOF $parent Assignment)
           (EQUAL $slot "lhs"))
      (THEN (PRINT stdout "Var %s defined at %s" $token $location))
      (ELSE (PRINT stdout "Var %s used at %s" $token $location)))))]
```

Using a conventional programming language to build even a simple tool such as this can require thousands of lines of code to parse the input, create the AST representation, resolve identifier references to derive the ASG from the AST, perform the traversal of all nodes in the ASG to identify the definitions and uses, and then print out the desired information. With Aria, however, the builders of testing or analysis tools need only be concerned with the application-specific aspects of the tools, leaving responsibility for

[1] An ASG is essentially an AST with embedded semantic information. In an AST, a reference to an entity is represented by an edge pointing to a simple leaf node that holds the name of the entity. In an ASG, a reference is represented by an edge pointing to the root of the (shared) subgraph in the ASG that represents the declaration of the entity.

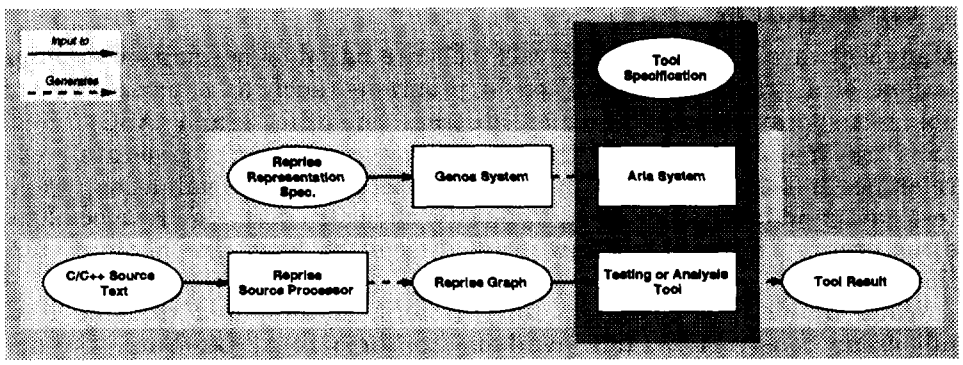

Fig. 1. Genoa, Reprise, Aria, and the testing and analysis tools.

the application-independent aspects (front-end processing, ASG generation, ASG traversal) to the Aria system.

We developed Aria by wedding a generic, language-independent tool generator, called Genoa, to a specific program representation, called Reprise. Genoa and Reprise were developed independently, and each has been described and compared to related work in previous papers [Devanbu 1992; Rosenblum and Wolf 1991]. Reprise provides a powerful and convenient ASG representation for C and C++ programs, but its use as the basis for testing and analysis tools was hindered by the lack of good tool construction aids. Genoa provides a terse, high-level specification framework that is intended to be generic across different program representations, but until its application to Reprise in the development of Aria, its utility in developing tools for complex manipulations of particular program representations was not demonstrated. As we describe in this article, our work on Aria presented the perfect opportunity to test out the approach to program representation embodied in Reprise with the approach to tool generation embodied in Genoa. In other words, Aria offers a proof of the concepts underlying Reprise and Genoa.

We begin in Section 2 with a description of how we constructed Aria using Genoa and Reprise. Section 3 describes the Aria specification language and presents some simple examples of Aria tool specifications. Sections 4, 5, and 6 present successively more sophisticated examples of Aria-developed tools—first one that computes the McCabe cyclomatic complexity metric, then one that computes approximate path conditions, and finally one that derives control dependence graphs directly from ASGs. We conclude in Section 7 with a summary of our results, a discussion of related work, and a sketch of our plans for future work.

2. THE CONSTRUCTION OF ARIA

Figure 1 puts the various elements discussed in this article into context by depicting the relationships among Genoa, Reprise, Aria, and the resulting testing and analysis tools. In the figure, rectangles denote processing

elements, and ovals denote data elements. A specification of the Reprise representation is fed into Genoa to produce an application generator, Aria, tailored to Reprise. Aria takes a specification of a Reprise graph manipulation and produces a tool that implements that manipulation. The generated tool itself takes particular Reprise graphs, which are produced from C and C++ source texts by the Reprise source processor, and computes results. Users of the testing and analysis tools would, of course, only need to know that they are supplying C or C++ source text to a tool that produces interesting results. Analogous construction aids for other languages and their program representations would be developed by feeding specifications of those representations to Genoa and developing the appropriate tool specifications.

2.1 Reprise

Builders of testing and analysis tools—that is, the users of Aria—need to understand the Reprise ASG in order to specify manipulations of the program representation. For purposes of this article, we give only a brief overview of the representation; details of Reprise can be found elsewhere [Rosenblum and Wolf 1991].

The conceptual model underlying Reprise is that of strongly typed expressions, where all semantic information is uniformly represented as the application of operators to arguments. For example, an if-statement is represented with the operator if, which is applied to three arguments—the first being an expression representing a condition, the second and third being expressions representing the then-part and else-part, respectively. As another example, the declaration of an object, such as an integer variable or an instance of a class (the C++ term for an abstract data type), is represented with the operator %object_decl, which is applied to three arguments, including one to represent the declared identifier, one to represent the type of the object, and one to represent the initial value of the object.

This simple conceptual model is realized in a graph data structure that employs two kinds of nodes and two kinds of directed edges. An *expression node* represents the application of an operator to arguments, while a *literal node* represents a literal appearing in a program, such as a number, a string, or the defining occurrence of an identifier. An *evaluation edge* corresponds to an edge in a traditional AST, while a *reference edge* is the semantic link (such as a link between a reference to a variable and the declaration of the variable) that turns the AST into an ASG. Thus, Reprise provides full semantic representation of C and C++ programs, with the underlying AST as a convenient subgraph for traversal.

2.2 Genii

In order to provide a tailorable specification framework, Genoa defines a standard representation structure for programs. This structure is essentially an abstract syntax tree. A tool called Genii is used to create a

```
If_Stmt: "ob->Kind() == RepriseNode::ExpNode
          && OperatorIs((ExpressionNode*)ob,\"if\")"
        < "ExpressionNode*" >
        {
          Cond:      an Expression < "ob->ArgumentNode(0)" >
          Then_Stmt: a Stmt < "ob->ArgumentNode(1)" >
          Else_Stmt: a Stmt < "ob->ArgumentNode(2)" >
        }
```

Fig. 2. Example production from the Genii specification for Reprise.

mapping from the basic Genoa AST structure to a particular program representation, such as the Reprise ASG. Thus, Aria-generated tools are specified in terms of manipulations of the Genoa AST, but are actually implemented—through the Genii mapping—as manipulations of the underlying Reprise representation.

Genii itself is based on application generator technology and therefore operates on a specification of the mapping. The Genii specification of Reprise consists of both declarative and operational elements. The declarative parts serve to describe a view of Reprise that is compatible with Genoa's AST nomenclature and style of tree manipulation. In particular, Reprise *operators* are given Genoa *node types*, and the numbered arguments of each Reprise operator are mapped to named *slots* in Genoa nodes. A slot can be thought of as a label on the Genoa AST edge from a parent node to one of its immediate children; the child is called the *filler* of the slot.

The operational parts of the Genii specification of Reprise serve to describe the runtime translation from a Reprise representation to a Genoa representation. These are given as actual code fragments to be inserted during the generation of the testing and analysis tools. The fragments describe how the tools are to extract and manipulate pieces of a Reprise graph during a traversal of the representation.

A Genii specification looks very much like a grammar specification as found in other language-based generation tools. Figure 2 presents an example of a "production" from the Genii specification for Reprise. The example gives the specification for the representation of if-statements. In this Genii specification, if-statements are given the node type If_Stmt, which is declared to have three slots. The slot Cond of the node type If_Stmt has a filler of node type Expression, which is used to represent the condition of the if-statement. The slots Then_Stmt and Else_Stmt are of node type Stmt and are used to represent the then- and else-parts of the if-statement, respectively.

As mentioned above, the node type and its slots are associated with code fragments, which appear in double quotes in the Genii specification. These code fragments are calls to the Reprise application program interface (API)[2] that perform such functions as node identification in, and edge

[2] The Reprise API is itself written in C++.

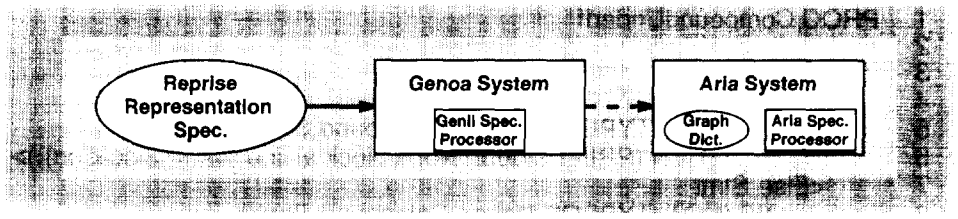

Fig. 3. Generation of the Aria system (detail from Figure 1).

traversal over, a Reprise graph. The code fragments manipulate a predefined object called ob, which represents the current Reprise node during a traversal; the notion of current node is described more fully in Section 4. For example, the code associated with the node type If_Stmt is the Reprise API code that determines whether or not if is the operator of the current Reprise expression node. The code associated with each slot of If_Stmt is the Reprise API code that retrieves the Reprise representation for that slot. Thus, the representation for slot Then_Stmt is retrieved as argument number 1 of the current Reprise expression node using the API function ArgumentNode. Note that, while not shown here, the code fragments in general must "collapse" the Reprise ASG into the simpler Genoa AST. In some cases this involves a sequence of Reprise API calls.

The Genii specification for Reprise comprises roughly 1500 lines of text, and its one-time development required roughly two person-months of (graduate student) effort. Remarkably, the developer of the Genii specification had no prior experience with C++, Genoa, or Reprise.

2.3 Aria

Once the Genii specification was developed, we generated the Aria system, as depicted in Figure 1. Figure 3 provides an expanded view of the generation of Aria. As shown in the figure, the Genii specification of the Reprise representation is fed into the Genii processor. The output of the Genii processor is a *graph dictionary* (which is akin to a data dictionary) containing the names of the node types and node slots plus the code fragments to be inserted by Aria into the tools that it generates. Thus, the Genii specification for Reprise results in a Reprise-specific instantiation of the Genoa specification framework—Aria. The graph dictionary in particular contains the vocabulary of the Aria specification language. We describe this language in the next section.

3. THE ARIA SPECIFICATION LANGUAGE

An Aria tool specification is a collection of one or more recursive procedures that manipulate a graph representation via *traversals*, *imperatives*, and *expressions*. These operators, together with recursion, provide sufficient expressive power to specify any tree manipulation, including those that rely on contextual information, such as nesting level of statements.

```
1   PROC CompoundUnderIf
2   {
3   [(? If_Stmt
4       <Then_Stmt
5           (IF (NOT (TYPEOF $token Compound_Stmt))
6               (THEN (PRINT stdout "Not a block at line %s\n" $location)))>
7       <Else_Stmt
8           (IF (NOT (TYPEOF $token Compound_Stmt))
9               (THEN (PRINT stdout "Not a block at line %s\n" $location)))>
10  )]
11  }
```

Fig. 4. Specification for the Aria-generated tool CompoundUnderIf.

At any point during the execution of a generated tool, the tool is operating on a *current node*. Traversals move the current node around the semantic graph as described in detail below; initially the current node is set to the root node of the graph. Imperatives perform variable assignment, printing, and other such functions. Expressions are formed from primitive values (e.g., the current node, the type of a node, or the location of a node in terms of a source line number), graph expressions (e.g., a reference to a particular child slot of another node), and other expressions (e.g., arithmetic expressions, list expressions, or procedure calls).

Traversals are central to the manipulation of a graph representation, and Aria provides four different kinds of traversals:

(1) ⟨slotname . . .⟩,[3] which retrieves the filler node of the current node's slotname and makes it the new current node;

(2) {nodetype . . .}, which iterates over a list of nodes of type nodetype (such as the list of statements associated with a block statement), making each one successively the current node;

(3) [. . .], which moves the current node successively over all the descendants (in the entire subtree) below the current node, using a depth-first search; and

(4) (? nodetype . . .), which tests whether the current node is of type nodetype.

These traversals are generic features of Genoa's specification framework. As described in Section 2, it is the collection of node types and slot names defined in the Genii specification for Reprise that instantiates Genoa's specification framework for Aria.

Figure 4 presents a simple example of an Aria specification, a specification of a tool called CompoundUnderIf. The purpose of CompoundUnderIf is to find all then- and else-parts of if-statements that are not compound block statements enclosed in curly braces. Such if-statement constructions are a frequent source of faults in C and C++ programs, since a forgotten pair of

[3] The notation ". . ." denotes elided uses of other (nested) operations and is not part of the Aria specification language.

curly braces for the sequence of statements forming a then- or else-part will cause all statements after the first statement in the sequence to be executed unconditionally outside of the if-statement. The test in line 3 confines the processing to all if-statements (i.e., all nodes of type If_Stmt). Lines 4 through 6 handle the then-part of each if-statement (contained in the slot Then_Stmt), while lines 7 through 9 handle the else-part (slot Else_Stmt). In each case, a diagnostic message is printed out if the filler of the slot is not of node type Compound_Stmt, which represents statement sequences enclosed in curly braces. Aria produces 392 lines of C code from this simple 11-line specification.

The Aria specification language offers the builder of a tool two main advantages: the traversal operations are terse, and they provide information and complexity hiding. A Reprise graph must use many different kinds of nodes to represent the many constructs of C and C++. Furthermore, the Reprise operators associated with these nodes vary in the number and meaning of the arguments they take and thus must be distinguished by different tests. Yet a single [. . .] traversal can be used to walk over all these different kinds of Reprise nodes, regardless of the operator associated with the node and the number and meaning of its arguments. Consider a global search idiom in Aria such as the following:

```
[ (? If_Stmt . . . (PRINT . . .))]
```

This compact idiom can be used to search everywhere in a file for occurrences of if-statements and then print out some piece of information about them. The Aria programmer simply has to specify that an If_Stmt node is desired, and then the proper traversals and tests on the Reprise graph will be automatically invoked to identify all such nodes. In the next three sections we present more sophisticated examples of Aria tool specifications that demonstrate these points.

4. AN ARIA TOOL TO COMPUTE THE MCCABE COMPLEXITY METRIC

McCabe proposed applying the graph-theoretic notion of *cyclomatic complexity* to control-flow graphs of software modules, in order to measure how difficult it is to comprehend, test, and maintain a module [McCabe 1976]. McCabe used empirical evidence to argue that a cyclomatic complexity of 10 or greater should be interpreted as an indication that a module is too complex to comprehend, test, or maintain. While the validity of this metric has long been the subject of considerable controversy, many software development organizations nevertheless make heavy use of the metric.

The cyclomatic complexity of a function is computed as one plus the number of conditional branches induced by the control-flow of the function. In C and C++, if-statements, while-loops, for-loops, and do-loops introduce conditional branching within the control-flow of a function. Similar to if-statements, the operator ?: is used to form conditional expressions and thus also introduces conditional branching. A switch-statement introduces conditional branching to each case inside the switch-statement. Finally, the

```
1   PROC Cyco
2   ROOT File;
3   {
4   LOCAL GNODE mccnt;
5   <Fdefs {Thingy
6     (? Func_Defn
7       (ASSIGN mccnt 1)
8       [(? If_Stmt (ASSIGN mccnt (+ 1 mccnt)))
9        (? While_Loop (ASSIGN mccnt (+ 1 mccnt)))
10       (? For_Loop (ASSIGN mccnt (+ 1 mccnt)))
11       (? Do_Loop (ASSIGN mccnt (+ 1 mccnt)))
12       (? Case (ASSIGN mccnt (+ 1 mccnt)))
13       (? LogAnd (ASSIGN mccnt (+ 1 mccnt)))
14       (? LogOr (ASSIGN mccnt (+ 1 mccnt)))
15       (? Question (ASSIGN mccnt (+ 1 mccnt)))]
16      (PRINT stdout "Function %s: %s\n" (SLOT F_Name $token) mccnt)
17  )}>}
```

Fig. 5. Aria specification of a tool to compute McCabe's cyclomatic complexity metric.

operators && (logical conjunction) and || (logical disjunction) introduce conditional branching, since the right-hand side of these operators is evaluated only if the value of the left-hand side does not by itself determine the value of the overall condition.

Figure 5 presents the complete Aria specification of our cyclomatic complexity tool, Cyco. The specification computes the metric for each function definition found in the underlying ASG. The metric is computed once per function by iterating over all function definitions found in a file.[4] Line 2 of the specification indicates that, at the start of processing, the current Reprise node is of node type File, which in Reprise represents the root of the graph corresponding to a file. Line 4 declares a local variable, mccnt, to keep track of the number of conditional branches found in a function. At line 5, the ⟨Fdefs . . .⟩ moves the current node from the root of the Reprise graph to the list of definitions ("thingys") in the file. Using the iteration {Thingy . . .}, each definition is visited in turn. Line 6 specifies that, of these definitions, we are interested in examining function definitions. Lines 7 through 16 detail the actions to be performed for each function definition. At line 7, the variable mccnt is initialized to one, where one is the cyclomatic complexity of a function containing no conditional branching. The value of mccnt is then incremented for each occurrence of one of the constructs described above. The occurrences are discovered through a depth-first search of the ASG representing the current function (lines 8 through 15). More specifically, Aria generates a single walk of the graph and queries for each of the eight node types of interest, rather than generating eight walks of the graph, one for each node type of interest. At

[4] The "file" is the C and C++ unit of compilation and visibility. An individual Reprise graph represents a single file, which encompasses any text inserted into the file by #include directives of the preprocessor.

```
1   int fun(int p1, int p2, int p3) {
2     int a;
3     int b;
4     int c;
5     a = p1;
6     while (a <= p2) {
7       if (a > p3)
8         a = 10;
9       else b = 5;
10    }
11    if (a > p3 || p2 < p3)
12      {
13        b = 5;
14        if (a <= b-3 && (p2 < p1 || p1 < p3))
15          return (0);
16        else c = 4;
17      }
18    else {
19      a = 5 - b;
20    }
21    a = p3 + p2;
22    return (a);
23  }
```

Fig. 6. A sample function.

line 16 the metric is printed out, along with the name of the function, which is the filler F_Name of the function slot.

Figure 6 presents a sample function named fun that we use to illustrate the operation of Cyco. We also use this function to illustrate the tools presented in the next two sections. When run on the representation for fun, Cyco computes the value 8 for the cyclomatic complexity.

Assuming that fun was in a file by itself, then the iteration construct on line 5 of the Aria specification would find only this one "thingy." And since fun is a function definition, the actions of lines 7 through 16 of the specification are carried out. These actions are quite straightforward, but we reiterate the point that only a single, depth-first walk of the representation is made to discover the one while-statement, three if-statements, and three logical operators contributing to the complexity measure of function fun.

Cyco consists of 925 lines of Aria-generated C code, yet the specification took only five minutes to develop.

5. AN ARIA TOOL TO COMPUTE APPROXIMATE PATH CONDITIONS

Many software projects have limited resources available for testing. One way to increase the effectiveness of a limited testing effort is to exploit the specific needs of the testers and the special properties of the application. In this section we describe a simple Aria-generated testing tool that was rapidly developed to help create unit test data for the function subprograms in a commercial C++ data processing application. The specification of this

tool is somewhat more complex than the specification for Cyco, illustrating the use of Aria's while-loop, list operators CAR and NEXT, and support for functional decomposition.

The testers of this application employ a rather pragmatic criterion to select test cases. The criterion is based on their expert knowledge about which statements are the critical ones to test. Specifically, they want to ensure the testing of statements that have customer-visible effects. Thus, the testers needed a tool that could help answer the following question: what are the test data that cause execution to reach a given statement in some function under test?

The key property of the application exploited by the tool is the fact that most of the functions exhibit a very restricted form of data dependence between inputs and outputs. In particular, the functions generally perform the following sequence of actions: read some data, validate the data, and perform a display or database update operation based on the data. Using traditional symbolic execution techniques, one could generate logical *path conditions*, each of which characterizes a path through a function and whose satisfiable instances represent the test inputs that exercise that path [Clarke and Richardson 1981]. However, symbolic execution techniques are computationally expensive. Fortunately, the nature of this application allows us to simply and efficiently compute useful approximations of a function's path conditions without resorting to symbolic execution.

We refer to the approximate path conditions as *syntactic path conditions* and automate their generation through our tool, which we call Synpatico. A syntactic path condition is a conjunction of the conditions found in the conditional statements along some path to a given line in a function. Synpatico generates all such conjunctions for the line, without simplification and using negations of individual conditions as necessary. The path conditions are "syntactic" in the sense that just one particular syntactic construct is examined (namely the conditions of conditional statements), and that semantic aspects of functions, such as data flow, are disregarded. Once the conditions are generated, a tester then formulates test data satisfying those conditions. It is of course possible to construct unsatisfiable path conditions for feasible paths by approximating in this manner. In practice, however, this has rarely been a problem for the testers of this application.

Figure 7 shows the output Synpatico generates for line 15 of the example function of Figure 6. The syntactic path conditions in that figure arise from the possible execution paths through the while-statement appearing on line 6 and the if-statements appearing on lines 11 and 14 of the function. For example, the second syntactic path condition appearing in Figure 7 corresponds to the path beginning at the function entry point, passing the while-statement, entering the then-part at line 12 (as a consequence of the left-hand disjunct in line 11 being true), and arriving in the then-part at line 15 (as a consequence of the left-hand conjunct and the left-hand disjunct of the right-hand conjunct in line 14 both being true).

```
********Begin Function Named fun ********
Line 15 condition: fun
                   && (a <= (b - 3)) Cond From: Line 14 in file "fun.c"
                   && !( (p2 < p1) ) Neg Cond From: Line 14 in file "fun.c"
                   && (p1 < p3) Cond From: Line 14 in file "fun.c"
                   && (a > p3) Cond From: Line 11 in file "fun.c"
                   && !( (a <= p2) ) Neg Cond From: Line 6 in file "fun.c"
Line 15 condition: fun
                   && (a <= (b - 3)) Cond From: Line 14 in file "fun.c"
                   && (p2 < p1) Cond From: Line 14 in file "fun.c"
                   && (a > p3) Cond From: Line 11 in file "fun.c"
                   && !( (a <= p2) ) Neg Cond From: Line 6 in file "fun.c"
Line 15 condition: fun
                   && (a <= (b - 3)) Cond From: Line 14 in file "fun.c"
                   && !( (p2 < p1) ) Neg Cond From: Line 14 in file "fun.c"
                   && (p1 < p3) Cond From: Line 14 in file "fun.c"
                   && !( (a > p3) ) Neg Cond From: Line 11 in file "fun.c"
                   && (p2 < p3) Cond From: Line 11 in file "fun.c"
                   && !( (a <= p2) ) Neg Cond From: Line 6 in file "fun.c"
Line 15 condition: fun
                   && (a <= (b - 3)) Cond From: Line 14 in file "fun.c"
                   && (p2 < p1) Cond From: Line 14 in file "fun.c"
                   && !( (a > p3) ) Neg Cond From: Line 11 in file "fun.c"
                   && (p2 < p3) Cond From: Line 11 in file "fun.c"
                   && !( (a <= p2) ) Neg Cond From: Line 6 in file "fun.c"
********End Function Named fun ********
```

Fig. 7. Syntactic path conditions generated by Synpatico for line 15 of the function in Figure 6.

Synpatico operates by traversing the Reprise graph of a function in statement execution order. As it traverses each statement, Synpatico keeps track of the set of evolving path conditions and outputs the set when the desired statement is reached. At branches in the execution path (such as at conditional statements) Synpatico traverses each branch in turn. Because the logical operators of C++ have short-circuit semantics (as described in Section 4), it is also necessary to traverse the individual disjuncts and conjuncts within compound conditions.

An excerpt from the Aria specification for Synpatico is shown in Figure 8. The figure shows the portion of the specification that handles an if-statement. The excerpt contains calls to three helper procedures, whose implementations are not shown. The procedures PosCond and NegCond return lists of formulae, each representing the possible ways to achieve a true value and a false value, respectively, for the if-statement's condition. The lists returned by PosCond and NegCond are stored in the variables tcond and fcond, respectively. These lists are manipulated using the Aria operators CAR and NEXT, in the obvious way. The then-part is processed beginning on line 4 by initializing the variable Start to the first condition stored in tcond. Line 5 makes the node representing the then-part of the if-statement the current node. The while-loop in lines 6 through 8 calls procedure HandleBasicBlock once for each condition stored in tcond. The

```
1 (? If_Stmt
2     <Cond (ASSIGN tcond (CALL PosCond $token))
3           (ASSIGN fcond (CALL NegCond $token))>
4     (ASSIGN Start (CAR tcond))
5     <Then_Stmt
6         (WHILE (NOT (NULL Start))
7                (CALL HandleBasicBlock Start)
8                (ASSIGN Start (NEXT Start)))>
9     (ASSIGN Start (CAR fcond))
10    <Else_Stmt
11        (WHILE (NOT (NULL Start))
12               (CALL HandleBasicBlock Start)
13               (ASSIGN Start (NEXT Start)))>)
```

Fig. 8. Excerpt from the Aria specification for Synpatico.

procedure is used to combine each such condition with the conditions generated for the conditional statements making up the then-part. The else-part of the if-statement is processed similarly in lines 9 through 13.

The entire Aria specification for Synpatico consists of approximately 450 lines and results in just over 10,000 lines of C code. The specification took roughly one week to develop.

6. AN ARIA TOOL TO DERIVE CONTROL DEPENDENCE GRAPHS

In this section we present a portion of the Aria specification for a tool called Tosca. Tosca produces a *control dependence graph* (CDG) of a program. A CDG is commonly used as one of the two main components of a graph-based formalism called a *program dependence graph* (PDG) [Ferrante et al. 1987].[5] PDGs have been used primarily for program optimization and parallelization. For instance, parallelizing Fortran compilers use PDGs to analyze dependencies among iterations of a loop in order to schedule the loop iterations for correct parallel execution. CDGs, both by themselves and as part of PDGs, have also been used in a variety of ways in testing and analysis [Harrold et al. 1993; Podgurski and Clarke 1990].

Constructing CDGs for arbitrary programs is a complex task; implementations of CDG tools can run into several thousand lines of code. Traditionally, a CDG is constructed by first deriving a control-flow graph (CFG) from an AST or ASG and then performing various transitive-closure and state maintenance operations to derive the CDG. This two-step implementation strategy directly follows from the standard CFG-based definition of CDGs [Ferrante et al. 1987]. In our formulation for Tosca, the CDG is derived directly from the Reprise ASG, eliminating the need for the CFG. Our algorithm is somewhat similar to that of Harrold, Malloy, and Rothermel [Harrold et al. 1993], in which a CDG is constructed during parsing and then adjusted to correct control dependence relationships for certain situations involving structured and unstructured transfers of control.

[5] The other main component of a PDG is the data dependence graph.

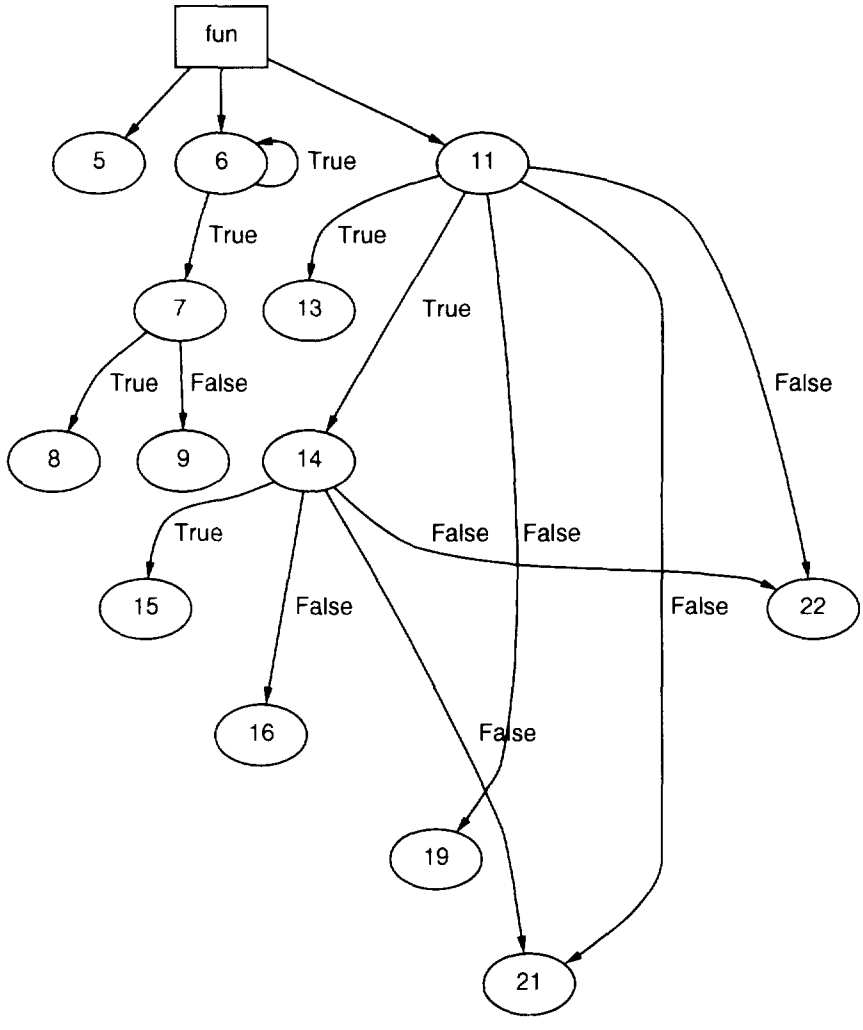

Fig. 9. Control dependence graph for the sample program of Figure 6.

While an algorithm for deriving a CDG directly from an ASG is relatively straightforward for structured programs, it is quite complex for unstructured programs. By definition, unstructured programs are those containing break-, continue-, or goto-statements, or with return-statements other than at the ends of functions. Unstructured programs exhibit quite intricate and sometimes counterintuitive dependencies that can be detected only through a significant amount of special-case analysis. Our initial version of Tosca handles unstructured programs containing if- and while-statements that enclose arbitrarily nested break-, continue-, and return-statements. We also make the standard assumption [Ferrante et al. 1987] that the program being analyzed does not contain unreachable statements.

Figure 9 shows the CDG that Tosca constructs from the sample program of Figure 6. The numbers on the nodes are statement line numbers from

the program text, and the edges indicate direct control dependencies. The direction of the edge should be interpreted to mean that the statement at the head of the edge is control dependent on the statement at the tail of the edge. Furthermore, the label on an edge is the value that the boolean condition in the statement at the tail of the edge must have to enable execution of the statement at the head of the edge. For example, the statement at line 8 is control dependent on the if-statement at line 7 and can be executed only if the condition in the if-statement evaluates to true.

6.1 Algorithm for CDG Generation

Before describing the Aria specification of Tosca, we briefly discuss the algorithm underlying the specification. In discussing this algorithm, we use several terms that need definition. A *control statement* is a statement that alters the control-flow of a program. Examples include if-statements, while-statements, and return-statements. A *compound statement* is a statement that contains other, nested statements. Examples include block statements, if-statements, and while-statements. In the case of while-statements, the while-statement is considered to contain itself. A *conditional control statement* is a control statement whose effect on the control-flow is subject to a logical condition; all other control statements are referred to as *unconditional control statements*. In C and C++, all conditional control statements are also compound statements, so we can consider these two kinds of statements as one.[6] We use the term *control point* to refer to any statement in a function on which the execution of other statements conditionally depends. For example, the head of a function is a control point, since without the function being called, none of the statements in its body would be executed. Likewise, a conditional control statement such as an if-statement is a control point for the statements within its then- and else-parts (as is a while-statement for the statements within its loop). In essence, our algorithm locates the control points in a program and generates edges, as well as labels for the edges, between the control points and the statements that directly depend on those control points.

The algorithm performs a traversal of the ASG that follows the control-flow of the program. As the traversal progresses, the algorithm maintains two *cursors*—that is, it remembers two positions within the ASG. One cursor is simply the current position of the traversal and is referred to as the *traversal cursor*. The other cursor is the current control point and is referred to as the *control cursor*. The initial position of both cursors is the head of the current function to be analyzed. The algorithm iteratively moves the traversal cursor along the control-flow paths of the function. When the traversal cursor encounters a conditional control statement, the algorithm recursively moves the control cursor to the position indicated by the traversal cursor, positions the traversal cursor at the beginning of the

[6] Ada is a language that provides some conditional control statements that are not compound statements. An example is the exit-when statement for loops.

statements nested within that control statement, and begins an iterative traversal of the nested statements.

The complication in the algorithm arises when the traversal encounters an unconditional control statement nested within a conditional control statement. We take as our example an if-statement that contains a nested return-statement. In this case, the statements that depend on the if-statement include not only the directly nested statements one would expect, but also the statements that follow the if-statement. The reason is that the execution of these statements, which are referred to as the *followers*, depends on whether the return-statement is executed, and the execution of the return-statement depends, ultimately, on the logical condition of the if-statement. This can be seen in Figure 9, where statements at lines 21 and 22 depend on the if-statement at line 11 only because of the presence of the (indirectly) nested return-statement at line 15. The figure illustrates a further complication, which is that the statements at lines 21 and 22 depend on the if-statement at line 14, as well.

The complication that is presented by unstructured control is the primary rationale for the traditional two-step algorithm that employs an intermediate CFG. Our algorithm instead addresses the problem in the following manner. First, the algorithm identifies the *unstructured control points*, which are the control points of statements that contain a (directly or indirectly) nested unconditional control statement. Second, the algorithm keeps track of the followers of an unstructured control point in a data structure called the *follow stack*; a stack is required because of the possibility of nested conditional control statements, each of which has its own followers. Third, when an unstructured control point is encountered during the traversal, the recursive analysis of the statement is provided with the current follow stack to aid in the analysis of dependencies and construction of edges.

6.2 Tosca Specification

Using the Aria specification language the CDG generation algorithm is implemented by two mutually recursive procedures. Excerpts from these procedures are shown in Figure 10. This portion of the specification handles if-statements that may possibly contain nested return-statements.

One procedure, StraightFlow, takes four arguments: a control point, a (possibly compound) statement, a dependence edge label, and a follow stack. StraightFlow iterates through the representation of "straight-line" code—that is, code that does not involve control statements—using a while-loop. When the procedure does encounter a control statement, it calls the second procedure shown in the figure, ControlFlow, to handle that statement. The mutual recursion of the two procedures comes about when ControlFlow in turn calls StraightFlow to handle the nested statements of the (compound) statement. ControlFlow takes two arguments, a control cursor and a follow stack.

Looking closer at StraightFlow, we see that it initializes the traversal cursor, Cursor, to the first statement to be analyzed and then uses a loop to

```
PROC StraightFlow
ARG  ControlPoint;
ARG  Statement;
ARG  Label;
ARG  FollowStack;
{
 LOCAL Cursor;
 (ASSIGN Cursor
    (CALL BlockFirst Statement))
 (WHILE (NOT (NULL Cursor))
    (CALL GenerateEdge
             ControlPoint
             Cursor
             Label)
    (IF (TYPEOF Cursor Return)
       (THEN
          (RETURN)))
    (IF (TYPEOF Cursor Control)
       (THEN
          (CALL ControlFlow
                   Cursor
                   Label
                   FollowStack)
          (IF (EQUAL 1
                 (CALL ContainsReturn
                          Cursor))
             (THEN
                (RETURN)))))
    (ASSIGN Cursor (NEXT Cursor))
 )
 (CALL StackUnfold
        ControlPoint
        Label
        FollowStack)
 ...
}
```

```
PROC ControlFlow
ARG  Cursor;
ARG  FollowStack;
{
 LOCAL GNODE NewStack;
 (? If_Stmt
    (IF (EQUAL 1
          (CALL ContainsReturn
                   Cursor))
       (THEN
          (ASSIGN NewStack
             (CONS (NEXT Cursor)
                   FollowStack)))
       (ELSE
          (ASSIGN NewStack NIL)))
    <Then_Stmt
       (CALL StraightFlow
             Cursor
             $token
             "True"
             NewStack)>
    <Else_Stmt
       (CALL StraightFlow
             Cursor
             $token
             "False"
             NewStack)>
 )
 ...
}
```

Fig. 10. Excerpts from the Aria specification for Tosca.

traverse the (possibly empty) sequence of statements. Within the loop, a call is made to the helper procedure GenerateEdge to construct a dependence edge from the node in the CDG representing the current control point to the node representing the statement at the current traversal cursor position. The next action taken by StraightFlow is determined by the type of statement at the current traversal cursor position. Basically, if the statement is a return-statement, then the procedure is finished, and itself returns. If instead the statement is a conditional control statement (i.e., type Control), then a call is made to ControlFlow to handle the statement. For all other types of statements, no further action is required, and the loop continues by moving the traversal cursor along to the next position. Once the loop terminates, it is possible for followers to still reside on the follow stack. These are processed by the procedure StackUnfold.

We next turn to ControlFlow, which determines the kind of control point under analysis and takes appropriate action. The actions for an if-statement are shown in the figure. The first action is to determine whether the condition of the if-statement is a structured or unstructured control point. If unstructured, then ControlFlow pushes the statements following the if-statement onto the follow stack. Otherwise, there are no followers that need to be considered. The filler nodes for the then- and else-part slots are

then examined in turn. The statements for each part are processed using a recursive call to StraightFlow.

Let us now trace through the construction of a portion of the CDG in Figure 9 for function fun, whose source is shown in Figure 6. StraightFlow is called with the head of the function as the control point, the function body as the statement, an empty label, and an empty follow stack. Although not evident in the specification for Tosca that is shown, the declarations in lines 2 through 4 of Figure 6 are ignored for purposes of CDG construction. The statement at line 5 of fun is not a control statement, so a dependence edge is simply generated to it from the function head. The next statement considered is at line 6, a while-statement. Since this is a control statement, ControlFlow is called to handle it after a dependence edge is generated between the function head and the statement. After the various dependence edges are generated for statements at lines 7 through 9, processing returns to consider the if-statement at line 11.

Here we have reached an unstructured control point, since there is a nested return-statement appearing at line 15. ControlFlow is called and discovers that the statement is an if-statement that is an unstructured control point. It therefore stacks the followers of the statement at line 11—namely, the statements at lines 21 and 22. Processing then moves to the then-part, which results in a recursive call back to StraightFlow with the argument ControlPoint referring to the statement at line 11. A dependence edge is generated between the control point at line 11 and the noncontrol statement at line 13. The statement at line 14 is, of course, an unstructured control point, which is handled by a recursive call back to ControlFlow. The follow stack in this call is initially the same stack that was formed during the previous call to ControlFlow to process the statement at line 11. Thus, any followers of the statement at line 14 would be added to the followers for the statement at line 11. It turns out, however, that there are no followers for the statement at line 14, and so the follow stack remains the same.

Processing of the then-part for the if-statement at line 14 results in a call to StraightFlow with the control point at line 14. This leads to the generation of an edge from the control point to the statement at line 15 and, since the statement is a return-statement, an immediate return (so to speak) back to the site of the call in ControlFlow. The else-part is then processed, from which an edge is generated from the control point to the statement at line 16. Since there are no more statements in the else-part, a call is made to StackUnfold. StackUnfold is responsible for generating the edges from the control point (i.e., the statement at line 14) to the appropriate statements in the follow stack—namely, the ones at lines 21 and 22.

Processing eventually returns to StraightFlow at the point where the traversal cursor is at the statement at line 14 and where all nested statements have been analyzed. Since this statement is one that contains a return-statement, processing immediately returns to ControlFlow at the point where the then-part of the if-statement at line 11 is being processed. The else-part is then processed in a similar manner, with processing eventually returning to StraightFlow at the point where all of the state-

ments from lines 11 through 20 have been analyzed. The control point is still at the statement at line 11 as the last two statements in the top-level traversal of the body of fun are processed. That processing results in the generation of dependence edges between the statement at line 11 and those last two statements at lines 21 and 22.

The specification of Tosca is about 400 lines in Aria's specification language. From this specification, Aria generates 9141 lines of C code, which does not include the code for the front-end processing provided by the Reprise source processor. The development of the specification took roughly two person-weeks, including the time to formulate the ASG-to-CDG algorithm. The use of compact, yet powerful, Aria idioms proved to be particularly invaluable in constructing such a complex tool with relatively modest effort. For example, the following essentially forms the body of function ContainsReturn used in Figure 10.

```
[ (? Return (RETURN 1) ) ]
```

In one line this function can detect the existence of a return-statement which is nested arbitrarily deeply within a conditional control statement.

7. CONCLUSION

We have described an application generator called Aria that generates testing and analysis tools for C and C++. Tools are generated by Aria from a terse, high-level, procedural specification. We illustrated the power of Aria's specification language with a number of tools, including a particularly complex tool that generates control dependence graphs directly from an ASG representation. With Aria, the tool builder obtains relief from two burdensome tasks: building a syntactic and semantic analyzer for the source language, and writing complex code for traversing the graph representation. Aria provides the former in its interface to the Reprise representation and the latter in its specification language. These are significant advantages for constructing tools to test and analyze programs written in C and C++, which are syntactically irregular and semantically complex languages. Overall, we feel that our experiences confirm the utility of the approach to program representation embodied in Reprise and the approach to application generation embodied in Genoa.

In contrast with Aria, previous application generators have been built around a monolithic architecture in which the front-end processing component and the tool generation component are custom designed from scratch and supplied as an integrated package (e.g., Gandalf [Habermann and Notkin 1986], Centaur [Borras et al. 1988], Refine [Reasoning Systems 1990], and Pan [Ballance et al. 1992]). A detailed discussion of this issue can be found in the previous paper on Genoa [Devanbu 1992]. As was predicted in the initial work on Genoa, the loosely coupled architecture of Aria saved us a great deal of work by allowing us to reuse an existing front-end processing component, Reprise, whose peculiarities were not specifically addressed in the design of Genoa. In addition, we believe that

Aria's terse traversal operations compare favorably with the more complex pattern- and rule-based language of Refine and with the Pascal-like language of Gandalf.

The approach most similar to ours is found in the ProDAG Toolset [Richardson et al. 1992]. ProDAG provides its users with extensive facilities for deriving arbitrary kinds of dependence graphs from a common program representation. These include a generic program representation data structure and a control-flow graph generator. ProDAG also provides a tool that generates the routines that build the dependence graphs from specifications of those dependence graphs. In essence, the ProDAG toolset is an application generator for the specialized testing and analysis domain of dependence graphs. Aria represents a generalization of the approach embodied in ProDAG. With Aria we seek to work not within any particular testing and analysis domain, such as dependence graphs, but more generally for any manipulation of a program representation that occurs as part of a testing or analysis tool. Thus, we have gone the next step beyond systems such as ProDAG, to where the application generators are themselves generated, in order to better serve a wider constituency of tool builders.

One weakness that we encountered in developing Aria was the lack of a rich collection of data structures. Genoa provides only lists, plus the stack of local variables inherent in the recursive calling structure of the Genoa specification framework. While in Tosca we were able to accommodate this limitation in choice of data structures, other testing and analysis tools may require more powerful data structures. Therefore, it may be necessary to expand the selection of data structures provided by Genoa.

A more general weakness in Aria is that it does not directly support modification of ASGs during analysis. This would be useful in tools such as program instrumentation systems that could benefit from an ability to annotate the basic ASG. While Aria supports calls to user-supplied functions that can perform arbitrary operations, a more direct mechanism for modifying ASGs would be useful.

In the future we intend to specify a number of other tools with Aria, including a full PDG generator (i.e., a generator of the combined control and data dependence graphs). The fact that we were able to specify a tool as complex as a CDG generator encourages us to further test the limits of the expressive power of Aria's specification language. We would also like to test the limits of our overall approach to tool generation. One interesting experiment would be to try to reproduce the ProDAG toolset. This would entail first generating an instantiation of Genoa for the representation used in ProDAG and then specifying the ProDAG tools in the resulting specification language.

ACKNOWLEDGMENTS

We thank David Kohr and Suneel Saigal, who played key roles in the implementation of Aria. We also thank Jeanne Ferrante for her contribu-

tions to our understanding of program dependence graphs. Finally, we thank Stuart Feldman for his helpful suggestions on improving the presentation of this work.

REFERENCES

BALLANCE, R., GRAHAM, S., AND VANTER, M.V.D. 1992. The Pan language-based editing system. *ACM Trans. Softw. Eng. Method. 1,* 1 (Jan.), 95–127.

BORRAS, P., CLÉMENT, D., DESPEYROUX, T., INCERPI, J., KAHN, G., LANG, B., AND PASCUAL, V. 1988. Centaur: The system. In *SIGSOFT '88: Proceedings of the 3rd Symposium on Software Development Environments.* ACM, New York, 14–24.

CLARKE, L. AND RICHARDSON, D. 1981. Symbolic evaluation methods for program analysis. In *Program Flow Analysis,* S. Muchnick and N. Jones, Eds. Prentice-Hall, Englewood Cliffs, N.J.

DEVANBU, P. 1992. GENOA—A customizable, language- and front-end-independent code analyzer. In *Proceedings of the 14th International Conference on Software Engineering.* IEEE Computer Society, Washington, D.C., 307–317.

FERRANTE, J., OTTENSTEIN, K., AND WARREN, J. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9,* 3 (Oct.), 319–349.

GRAY, R., HEURING, V., LEVI, S., SLOANE, A., AND WAITE, W. 1992. Eli: A complete, flexible compiler construction system. *Commun. ACM 35,* 2 (Feb.), 121–131.

HABERMANN, N. AND NOTKIN, D. 1986. Gandalf: Software development environments. *IEEE Trans. Softw. Eng. SE-12,* 12 (Dec.), 1117–1127.

HARROLD, M. J., MALLOY, B., AND ROTHERMEL, G. 1993. Efficient construction of program dependence graphs. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA).* ACM, New York, 160–170.

MCCABE, T. 1976. A complexity measure. *IEEE Trans. Softw. Eng. 2,* 4 (Dec.), 308–320.

PODGURSKI, A. AND CLARKE, L. 1990. A formal model of program dependencies and its implications for software testing, debugging, and maintenance. *IEEE Trans. Softw. Eng. 16,* 9 (Sept.), 965–979.

REASONING SYSTEMS. 1990. *REFINE 3.0 User's Guide.* Reasoning Systems, Inc., Palo Alto, Calif.

REPS, T. AND TEITELBAUM, T. 1984. The synthesizer generator. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* ACM, New York, 42–48.

RICHARDSON, D., O'MALLEY, T., MOORE, C., AND AHA, S. 1992. Developing and integrating ProDAG in the Arcadia environment. In *SIGSOFT '92: Proceedings of the 5th Symposium on Software Development Environments.* ACM, New York, 109–119.

ROSENBLUM, D. AND WOLF, A. 1991. Representing semantically analyzed C++ code with Reprise. In *Proceedings of the 3rd C++ Technical Conference.* USENIX Assoc., Berkeley, Calif., 119–134.

Received August 1995; revised November 1995; accepted December 1995