Automated Construction of Testing and Analysis Tools

Research Paper

Premkumar T. Devanbu[†] David S. Rosenblum[†] Alexander L. Wolf[‡]

[†]Software and Systems Research Laboratory
AT&T Bell Laboratories
Murray Hill, NJ 07974 USA
{prem,dsr}@research.att.com

[‡]Department of Computer Science
University of Colorado
Boulder, CO 80309 USA
alw@cs.colorado.edu

Abstract

Many software testing and analysis tools manipulate graph representations of programs, such as abstract syntax trees or abstract semantics graphs. Hand-crafting 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 paper we report on our experiences in developing 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 tool that derives control dependence graphs directly from Reprise abstract semantics graphs.

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., Gandalf [6], the Cornell Synthesizer Generator [10], Centaur [2], Refine [9], Pan [1], and Eli [5]), the same cannot be said for builders of testing and analysis tools. Typically, testing and analysis tools are individually hand-crafted 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 derivations of definition/use chains or program dependence graphs, 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 paper 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 complex 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 prints out information about 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 "\n Var %s defined at %s"
                   $token $location))
      (ELSE (PRINT stdout "\n 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

[1]An ASG is essentially an AST with embedded semantic information. Specifically, 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.

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 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 [3, 12]. 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 paper, 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.

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. Section 4 demonstrates the utility of Aria and, thereby, the power of our approach, by showing Aria's use in the development of a particularly complex tool, one that derives control dependence graphs directly from ASGs. We conclude in Section 5 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 paper 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.

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 paper, we give only a brief overview of the representation; details of Reprise can be found elsewhere [12]. 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 an expression representing a condition, the second and third an expression 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 and another to represent the type 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 an identifier, a number, or a string. 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.

The mechanism that Genoa provides for binding its generic, language-independent specification framework to a particular program representation, such as Reprise, is called Genii. Genii itself is based on generation technology. To create Aria, it was first necessary to develop a Genii specification of the Reprise representation, mapping Genoa's generic, abstract model of ASGs to the Reprise representation data structure. The Genii specification performs two main functions. First, the specification describes a view of Reprise that is compatible with Genoa's nomenclature and style for

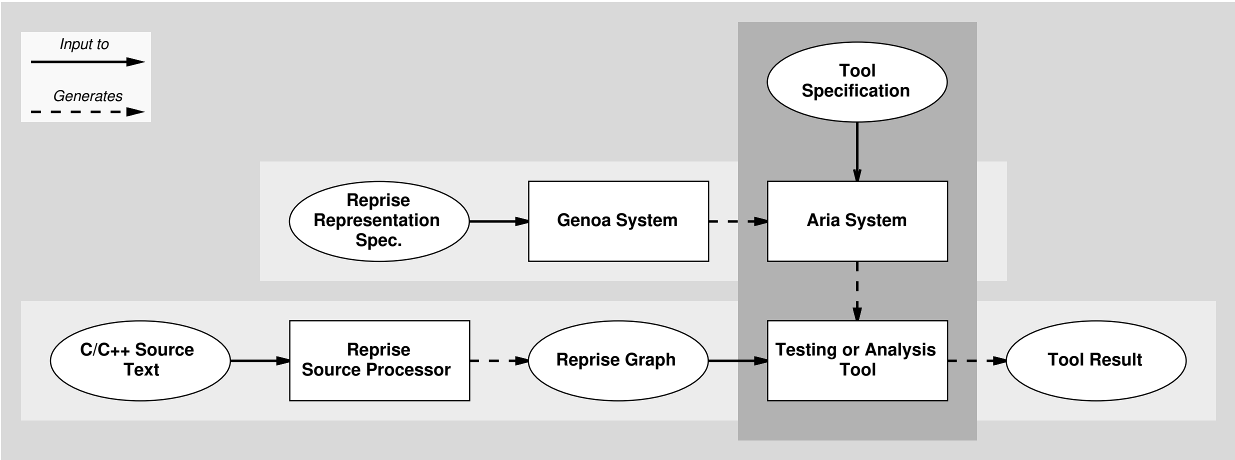

Figure 1: Genoa, Reprise, Aria, and the Testing and Analysis Tools.

manipulating graph-based program representations. In particular, Reprise operators are mapped to Genoa *node types*, and the arguments to each Reprise operator are mapped to named *node slots*. A slot is a label on the edge from a parent node to one of its immediate children; the child at the other end of the edge is called the *filler* of the slot. The second function of the Genii specification is to provide code fragments to be used during the generation of the testing and analysis tools. These fragments describe how the tools are to extract and manipulate pieces of a Reprise graph during a traversal of the representation. Thus, these code fragments implement the mapping between Genoa's abstract model of ASGs and the underlying physical Reprise representation. 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; the developer of the Genii specification had no prior experience with C++, Genoa, or Reprise.

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 illustrates the specification of the Genii mapping for the Reprise representation of if-statements. In the Genii specification, if-statements are given the node type `If_Stmt`, which has three slots. The slot `Cond` of the node type `If_Stmt` has a filler of node type `Expression`, which represents the condition of the if-statement. The slots `Then_Stmt` and `Else_Stmt` are of node type `Stmt` and 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. These code fragments are calls to the Reprise programmatic interface that perform such functions as node identification in and edge traversal over a Reprise graph. The code fragments manipulate a predefined object called `ob`, which represents the *current node* during a traversal; the notion of current node is described more fully in the next section. Thus, for example, the code associated with the node type `If_Stmt` would be executed to determine whether or not `%if` is the operator of a Reprise expression node. If the test succeeds, then the code associated with the slot `Then_Stmt` could be executed to retrieve the Reprise representation of the then-part of the statement.

Once the Genii specification was developed, we generated the Aria system, as depicted in Figure 1. Figure 3 provides an expanded view of the Aria generation. 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 used during tool generation. In this way the Genii specification for Reprise (together with the graph dictionary that is created by Genii) describes a Reprise-specific instantiation of the Genoa specification framework—Aria. The graph dictionary contains the vocabulary for the Aria specification language. We describe this language in the next section.

```
If_Stmt:  "ob->Kind() == RepriseNode::ExpNode && OperatorIs((ExpressionNode*)ob, \"%if\")"
          < "ExpressionNode*" >
          {
            Cond:       an Expression < "ob->ArgumentNode(1)" >
            Then_Stmt:  a Stmt < "ob->ArgumentNode(2)" >
            Else_Stmt:  a Stmt < "ob->ArgumentNode(3)" >
          }
```

Figure 2: Example Production from the Genii Specification for Reprise.

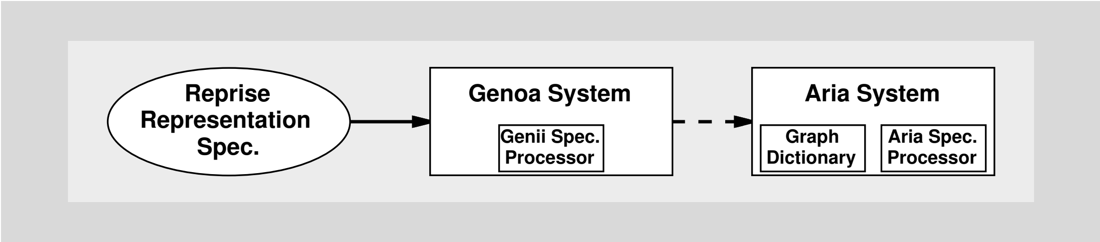

Figure 3: Generation of the Aria System (Detail from Figure 1).

3 The Aria Specification Language

The Aria specification language is a Reprise-specific instantiation of the generic specification framework provided by Genoa. A tool specification is a collection of one or more recursive procedures that manipulate a graph representation via *traversals*, *imperatives*, and *expressions*. 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 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, and the location of a node in terms of a source line number), graph expressions (e.g., reference to a particular child slot of another node), and other expressions (e.g., arithmetic expressions, list expressions, and procedure calls).

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

1. `<slotname ...>`,[2] which moves the current node to the filler node of `slotname`;

2. `{nodetype ...}`, which iterates over a list of nodes of type `nodetype`, 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; and

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

[2]The notation "..." denotes uses of other (nested) operations.

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 and that is used when specifying traversals.

As an example of an Aria specification, Figure 4 presents the specification for a simple Aria-generated tool, `FindPtrArg`. `FindPtrArg` finds all the instances where a function call contains as one of its arguments a pointer expression (i.e., an expression of the form `&foo`, which uses the C "address-of" operator `&`). As shown in the figure, an Aria specification comprises one or more procedure declarations; in this example the name of the procedure is declared in line 1. Line 2 declares a local variable called `fname` that is used as a temporary variable inside the procedure.

The procedure begins in line 3 by traversing to the slot `globals` of the `File` node. The filler of this slot is a list of nodes of type `Declaration`, which represent the variable, type, and function declarations contained

```
1    PROC FindPtrArg
2    {LOCAL fname;
3     <globals {Declaration
4              <identifier (ASSIGN fname $token)>
5              [(?FunCall <CallArgs {Expression
6                                    <ExprOp (?AddrOf
7                                             (PRINT stdout
8                                              "Pointer Arg used within %s at %s\n"
9                                              fname $location))>
10                                  }
11                        >
12             )]}
13   >}
```

Figure 4: Specification for the Aria-generated Tool `FindPtrArg`.

in the source file. In line 3 the procedure begins an iteration over each such declaration; this iteration extends to the end of the procedure. In line 4 it then saves the name of the entity being declared in the temporary variable `fname`. The name is a literal contained in slot `identifier`, whose value is contained in the predefined variable `$token`. Starting at line 5 the procedure examines all the nodes that are structurally below the root node of the declaration, searching for nodes that represent function calls. For each such function call that is found, the procedure iterates over all the declarations of arguments to the function. In line 6 each declaration is tested to see if it involves a call to the address-of operator `&`. For every such successful test, the procedure, in lines 7 through 9, prints out the name of the enclosing declaration (which was saved in `fname`) and the line number of the pointer expression. When processed through Aria, this simple 13-line specification generates 431 lines of C code that would otherwise have to be written by the tool builder, and this does not even include the code for the front-end processing provided by the Reprise source processor.

The Aria specification language offers the builder of a tool two main advantages: the traversal operations are terse, and they provide information/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 traversal and test on the Reprise graph will be implicitly invoked to identify all such nodes. In the next section we present a more complex example of an Aria tool specification that demonstrates these points.

4 An Aria-generated Tool to Derive Control Dependence Graphs

In this section we illustrate our approach to building testing and analysis tools by presenting 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 well-known graph-based formalism called a *program dependence graph* (PDG) [4].[3] 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. For examples, see Podgurski and Clarke [8], Harrold et al. [7], and the citations contained therein.

Constructing CDGs for arbitrary programs is a complex task; implementations of CDG tools can run

---

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

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 [4]. In our formulation for Tosca, the CDG is derived directly from the Reprise ASG, eliminating the need for the CFG.

While an algorithm for deriving a CDG directly from an ASG is relatively straightforward for structured programs, it is quite complex for unstructured programs—that is, programs with 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 have within them arbitrarily nested break, continue, and return statements. We also make the standard assumption [4] that the program being analyzed does not contain unreachable statements.

Figure 5 shows an example input program and a depiction of the CDG that Tosca constructs from this input. 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 dependent on the statement at the tail of the edge. Further, 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.

An excerpt from the Aria specification for Tosca is shown in Figure 6. This portion of the specification handles if-statements that may possibly contain nested return statements.

Before describing the excerpt itself, we briefly discuss the algorithm that is embodied in the Aria specification for Tosca. 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.[4] 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 roughly 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 a function under analysis. The algorithm iteratively moves the traversal cursor along the control flow path 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 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 we refer to as the *followers*, depends on whether the return statement is executed, and the execution of the return statement depends, ultimately, on the logical

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

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

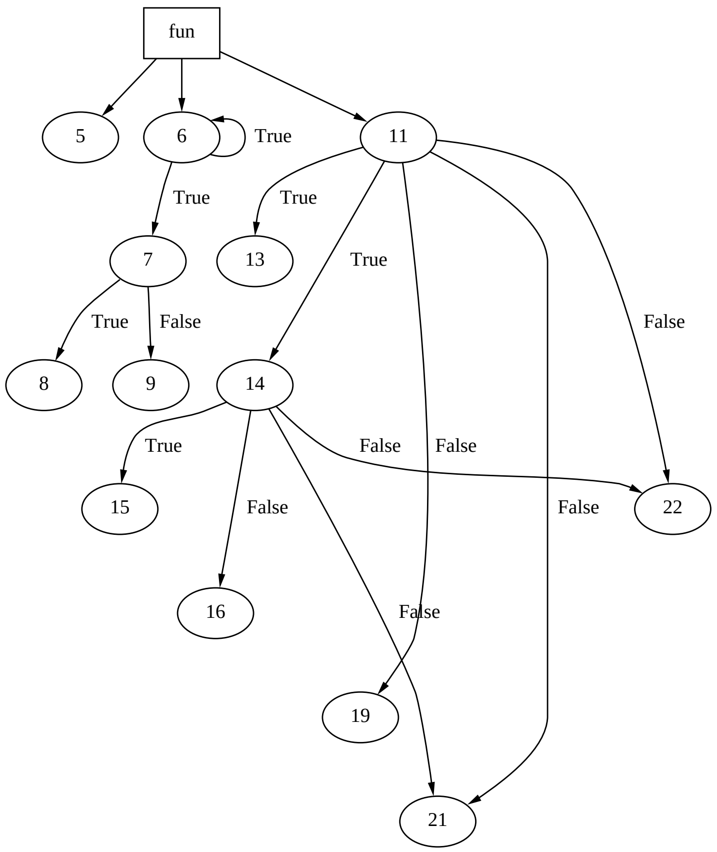

Figure 5: A Sample Program and its Control Dependence Graph.

condition of the if-statement. This can be seen in Figure 5, 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 that refer to conditional control statements containing 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.

Using the Aria specification language the algorithm is implemented by two mutually recursive procedures, as shown in Figure 6. In the figure, Aria keywords are written in all uppercase letters. 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 a compound statement. `ControlFlow` takes two arguments, a control cursor and a follow stack.

```
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
           (CALL ContainsReturn Cursor)
           1)
           (THEN (RETURN)))))
    (ASSIGN Cursor (NEXT Cursor))
 )
 (CALL StackUnfold
           ControlPoint
           Label
           FollowStack)
 . . .
}
```

```
PROC ControlFlow
ARG Cursor;
ARG FollowStack;
{
 LOCAL GNODE NewStack;
 (?If_Stmt
     (IF (EQUAL
            (CALL ContainsReturn Cursor)
            1))
         (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)>
 )
 . . .
}
```

Figure 6: Excerpt from the Aria Specification for Tosca.

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 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 if-statement is a structured or unstructured control point. If unstructured, then it 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 for function fun in Figure 5. 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 code for Tosca that is shown, the declarations in lines 2 through 4 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, and so a call is again made to `ControlFlow`. `ControlFlow` discovers that the statement is an if-statement and, in fact, 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 non-control statement at line 13. The statement at line 14 is, of course, an unstructured control point, which results in a recursive call back to `ControlFlow`. The follow stack in this call is 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 will 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, the while-statement terminates and 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 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 statements 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 new 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 6.

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

In one line this function can detect the existence of return statements nested arbitrarily deep within a conditional control statement. Overall, we feel that this experience confirms the utility of the approach to program representation embodied in Reprise and the approach to application generation embodied in Genoa.

5 Conclusion

We have described an application generator called Aria that generates testing and analysis tools for C and C++. Aria was created by combining two independently developed systems, namely an ASG representation for C and C++ called Reprise, and Genoa, itself an application generator. Tools are generated by Aria from a terse, high-level, procedural specification. We illustrated the power of Aria's specification language with a particularly complex tool, one 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 the 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.

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 [6], Centaur [2], Refine [9] and Pan [1]). A detailed discussion of this issue can be found in the previous paper on Genoa [3]. 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 (namely Reprise). 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 [11]. 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 significant expansion on 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.

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 Dave Kohr and Suneel Saigal, who played key roles in the implementation of Aria. We also thank Jeanne Ferrante for her contributions to our understanding of program dependence graphs. Finally, we thank Stu Feldman and the anonymous referees for their helpful suggestions on improving the presentation of this work.

REFERENCES

[1] R.A. Ballance, S.L. Graham, and M.L. Van De Vanter. The Pan Language-Based Editing System For Integrated Development Environments. In *SIGSOFT '90: Proceedings of the Fourth Symposium on Software Development Environments*, pages 77–93. ACM SIGSOFT, December 1990.

[2] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: The System. In *SIGSOFT '88: Proceedings of the Third Symposium on Software Development Environments*, pages 14–24. ACM SIGSOFT, November 1988.

[3] P.T. Devanbu. GENOA—A Customizable, Language- and Front-end-independent Code Analyzer. In *Proceedings of the 14th International Conference on Software Engineering*, pages 307–317. IEEE Computer Society, May 1992.

[4] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, October 1987.

[5] R.W. Gray, V.P. Heuring, S.P. Levi, A.M. Sloane, and W.M. Waite. Eli: A Complete, Flexible Compiler Construction System. *Communications of the ACM*, 35(2):121–131, February 1992.

[6] N. Habermann and D. Notkin. Gandalf: Software Development Environments. *IEEE Transactions on Software Engineering*, SE-12(12):1117–1127, December 1986.

[7] M.J. Harrold, B. Malloy, and G. Rothermel. Efficient Construction of Program Dependence Graphs. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, pages 160–170. ACM SIGSOFT, June 1993.

[8] A. Podgurski and L.A. Clarke. A Formal Model of Program Dependencies and its Implications for Software Testing, Debugging, and Maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.

[9] The Reasoning Systems Corporation, Palo Alto, California. *Refine Users Manual.*

[10] T. Reps and T. Teitelbaum. The Synthesizer Generator. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 42–48. ACM SIGSOFT, April 1984.

[11] D.J. Richardson, T.O. O'Malley, C.T. Moore, and S.L. Aha. Developing and Integrating ProDAG in the Arcadia Environment. In *SIGSOFT '92: Proceedings of the Fifth Symposium on Software Development Environments*, pages 109–119. ACM SIGSOFT, December 1992.

[12] D.S. Rosenblum and A.L. Wolf. Representing Semantically Analyzed C++ Code with Reprise. In *Proceedings of the Third C++ Technical Conference*, pages 119–134. USENIX Association, April 1991.