

A Toolsuite for Testing Real-Time Ada Applications*

Yann-Hang Lee, YoungJoon Byun, Ji Xiao, Okehee Goh
Computer and Information Science and Engineering
University of Florida
{yhlee, ybyun, jxiao, ogoh}@cise.ufl.edu

W. Eric Wong
Applied Research
Telcordia Technologies
ewong@research.telcor
dia.com

Alice Lee
Johnson Space Center
NASA
email[*FIXME?*]

Abstract

Generally, software testing is considered as the most expensive phase in software development. To reduce the development cost and improve productivity and quality, many tools are suggested and used. One example is Telcordia's *cSuds*** which not only analyze test coverage for C programs, but also apply the coverage information for test case selection and optimization.

In this paper, we will describe our development effort on an Ada program instrumentation environment, ADA-PINE, for test coverage analysis of real-time Ada programs. Following the similar approach as *cSuds*, ADA-PINE analyzes an Ada program and generates program flow graphs based on control flow and data flow of the program. It also performs instrumentation of the code and collects execution traces. The tool can be integrated with *cSuds*, thus allows developers and testers to identify the program areas that require further testing, to find the minimal covering test sets, and to select the optimal cases for regression tests.

1. Introduction

Software testing is one of the most expensive phases of the software life cycle. Hence, it is very important to provide a solution, supported by tools, which not only can reduce the cost but also improve the quality. One example is Telcordia's Software Visualization and Analysis Toolsuite (known as χ Suds) [3]. However, this toolsuite only supports C and C++, but not Ada, a widely used real-time programming language in the defense industry as well as many aeronautic applications such as those in the space shuttle and space station [2][6].

In this paper, we will discuss our research on developing an Ada program instrumentation

environment, ADA-PINE, for test analysis of real-time Ada programs. Based on the program analysis, ADA-PINE generates control flow and data dependence graphs, instruments the programs being tested, and collects the execution traces. The tool can help developers and testers in many ways such as determining how well the software has been tested and displaying the code that has not been executed. It can also conduct test set minimization with respect to code coverage and select effective fault-revealing regression tests. Furthermore, ADA-PINE can be used as a framework for analyzing and verifying concurrent execution models and different timing criteria for real-time systems.

2. Software Test Coverage Analysis

Software test coverage measures how adequately a set of test cases has tested the software with respect to certain criteria [5]. It is required, in many safety-critical software systems, that the test can only become adequate with respect to a given criterion if all of the attributes identified by the criterion are exercised at least once. To analyze test coverage, we need to gain the information of program execution paths on the internal structure of the source code. It identifies program constructs that may be exercised during program execution and determines which of these constructs are in fact exercised by test cases. The structural coverage testing has many coverage criteria such as block coverage, decision coverage, c-use coverage, p-use coverage, all-paths coverage, all-du-paths coverage, etc [4][8][9].

The goal of the project is to develop a test environment with which various characteristics of Ada programs for real-time systems can be measured and tested. As the first step, ADA-PINE identifies program constructs that may be exercised during program execution and determines which of these constructs are in fact exercised by test cases. Thus, it can measure the block, decision, c-use, p-use, and all-uses coverage.

A *basic block*, or simply a *block*, is a code sequence in which flow of control enters at the beginning and

* This work is supported by NASA Johnson Space Center with collaboration by Telcordia Technologies, Inc.

** *cSuds* is a registered trademark of Telcordia Technologies, Inc.

```

WITH Ada.Integer_Text_IO;
WITH Ada.Text_IO;
PROCEDURE Sample IS
  Count, Product, Num1 : Integer;
BEGIN -- Sample
  Ada.Text_IO.Put ("Enter an Integer = "); →1
  Ada.Integer_Text_IO.Get (Num1); → 2
  Product := Num1; → 3
  Count := 1; → 4
  WHILE Num1 /= 0 AND THEN Count < 10 LOOP → 5
    IF Product > 0 THEN → 6
      Product := Count * Num1; → 7
    ELSE
      Product := Count * (-Num1); → 8
    END IF;
    Count := Count + 1; → 9
  END LOOP;
  Ada.Integer_Text_IO.Put (Product); → 10
END Sample;

```

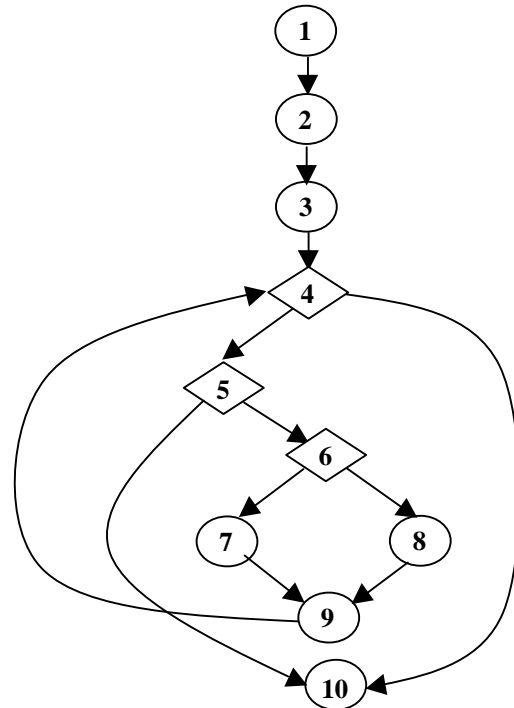


Figure 1. A sample Ada program and its control flow graph

leaves at the end without halt or possibility of branching except at the end. *Block coverage*, sometimes called *statement coverage*, of a set of tests on a program is the ratio of the number of blocks executed by the tests to the total number of blocks.

A *decision* is a pair of blocks, denoted (n_a, n_b) , which shows a control transfer from a block n_a to another block n_b . The *decision coverage* of a set of tests is the ratio of the number of decisions covered by the tests to the total number of decisions in the program. Block and decision coverage are based on the control flow of a program that can be represented in a *control flow graph*, with a node for a block and an edge for control transfer from one node to another node. Figure 1 shows a sample program and its control flow graph. As the figure shows, an Ada program may be considered to be a sequence of blocks.

For the analysis of block coverage testing, ADA-PINE divides a program into a sequence of blocks, and then creates edges among them. A block may have more than one statement if there is no branching among statements. A statement may contain multiple blocks if there is a control transfer within the statement. For example, an assignment statement with control change constructs in right hand side can be divided into two blocks, one for right hand side and another for the assignment of the result of right hand side into left hand side. An expression may also contain multiple blocks if there is branching implied in the expression. Ada 95 has many

constructs which change the control of the program. These constructs include short-circuit forms (*and then* and *or else*), select statement such as if statement and case statement, loop (basic loop, while, for), goto, exit, return, and subprogram call statements. The short-circuit control forms are specific to Ada and they have same results as the corresponding logical operators, but the left operand is always evaluated first, and depending on the result of the left operand the right operand may not be evaluated as the decision (n_4, n_{10}) at the figure1. A subprogram call is considered a possible control flow change since it may end the flow of control by exceptional termination as block1 at the figure1.

In addition to the control flow analysis, data flow analysis can be used to show how thoroughly the associations between the definition of a variable and its uses are exercised during program test. The analysis is done with *data flow graph* that is defined on control flow graph by decorating each node with variable usage information. A variable occurrence in a program can be classified as definition (*def*), computational-use (*c-use*), and predicate-use (*p-use*). For example, node 3 at the figure1 has *def* of variable *Product* and *c-use* of *Num1*. A *path* in a data flow graph is a finite sequence of nodes (n_1, \dots, n_k) , $k \geq 2$, such that there is an edge from n_i to n_{i+1} for $i = 1, 2, \dots, k-1$. A path is *simple* if all nodes, except possibly the first and last, are distinct and a path is *loop-free* if all nodes are distinct [8].

A *c-use paths* of a variable x in terms of *def* node n_a in which x is defined in this node and *c-use* node n_b is the set of simple paths from node n_a to n_b with no redefinition of x in the path. A *c-use* of a variable regarding a definition is covered if at least one of the *c-use* paths is executed when a test is run. The *c-uses coverage* of a set of tests is the percentage of *c-uses* covered by the tests.

A *p-use paths* of a variable x in terms of *def* node n_a and decision (n_b, n_c) is the set of loop-free paths from node n_a to (n_b, n_c) with no redefinition of x in the path, where n_b has x in a predicate and n_c is a subsequence block of n_b . A *p-use* of a variable regarding a definition is covered if at least one of the paths is executed when a test is run. The *p-use coverage* of a set of tests is the fraction of *p-uses* covered by the tests. *All-uses coverage* is the sum of *c-use* and *p-use* coverage. The idea behind *c-use* and *p-use* coverage is that program paths along which variables are defined and used should be covered.

3. Implementation of ADA-PINE

3.1. Architecture of ADA-PINE

ADA-PINE takes the similar approach as the Telcordia's χ Suds . ADA-PINE first analyzes source code and generates flow graph for the code. It traverses the graph and determines where code coverage

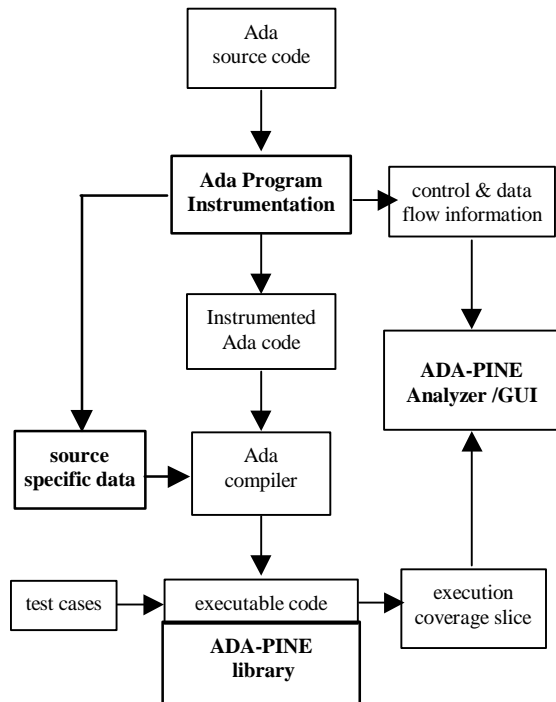


Figure 2. High level architecture of ADA-PINE

measurements should be made. For example, an instrumentation probe makes a measurement in each basic block. At those measurement points, the tool inserts an instrumentation code, i.e., a test probe, which will record dynamic behavior of the program. During the test process, the probes confirm that a test actually traversed the intended block.

Figure 2 shows the general architecture of ADA-PINE. It is composed of Ada program instrumentation, source-specific data structure, ADA-PINE library, analyzer and GUI. To perform the test coverage analysis for Ada, the program instrumentation part reads the source and generates an instrumented Ada code. The instrumentation process can be thought of a language translation. It generates the control and data flow information, including static information of source code such as file information, subprogram name and position, basic blocks features, decision information, and variable define-use information. It also generates source-specific data structure, which will be used by ADA-PINE library. After instrumentation, a tester can compile the instrumented code and exercise with his/her test cases. As the code is executed, the instrumented code generates dynamic information about which blocks, decisions, and paths are being exercised. ADA-PINE library does the work and records the runtime behavior of instrumented code and provides trace information for the dynamic behavior. The coverage analyzer then uses the collected information for analysis, display, or printing via the graphical user interface.

3.2. Ada instrumentation processor

Ada instrumentation processor is a kind of translator that carries out lexical analysis, syntax analysis, semantic analysis, flow analysis, instrumentation and code generation [1]. Figure 3 shows the implementation of Ada instrumentation processor. We assume that the source code to be instrumented is syntactically and semantically correct because the code has already passed the compilation phase for testing. The processor, therefore, performs minimum syntax and semantic analysis only for test analysis. Lexical analyzer scans the source code and produces a sequence of tokens that the syntax analyzer uses for syntax analysis. It also passes the token position in source code and text string of identifier and literal.

The syntax analyzer, generated by YACC [7], builds an abstract syntax tree of the source code based on the Ada syntax. The tree node has its node type with respect to a syntax rule and symbol information space, which is actually filled during semantic analysis.

The main function of semantic analysis is resolution of symbols such as variable name, constant name and subprogram name in source code. To classify the symbols, semantic analyzer traverses the syntax tree and identifies the meaning of symbol based on its context on source. It also identifies type information of each symbol and decorates the syntax tree with the symbol

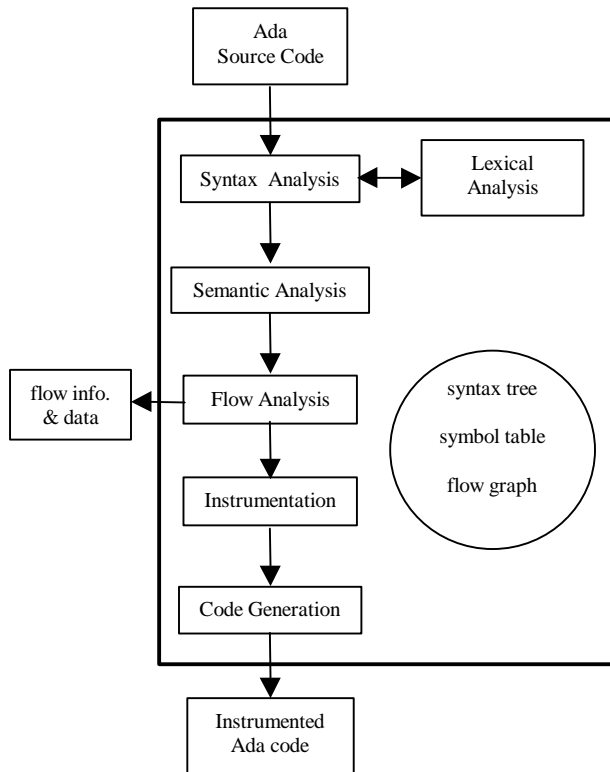


Figure 3. Ada instrumentation processor

information. The syntax tree nodes corresponding to compilation, subprogram body, and nested block maintain symbol tables including all symbols defined under the nodes, which are used to handle the scope of declaration. So compilation symbol table has information on all global symbols.

ADA-PINE resolves the symbols defined at the outside packages in separation compilation environment. In C, *cSuds* starts from the preprocessed source code where all name information is included in the file, but in Ada, the external symbols can be used by *with* statement. Semantic analyzer also handles the subprogram overloading, renaming, and *string literal* function name.

The flow analysis is divided into control flow analysis and data flow analysis. As described at section 2, block and decision represent program control. So the flow analyzer divides the source code into blocks and decisions at first. The flow analyzer generates a flow graph representing block and decision for each

subprogram by traversing the syntax tree. A node of flow graph indicates a basic block by pointing to a start node and an end node of syntax tree, which corresponds to start and end positions of source text, and an edge of flow graph shows a decision. The flow analyzer must recognize Ada 95 constructs changing the control of program. In addition to the control flow information, flow analyzer adds the data flow information to the graph by identifying the symbol's usage (eg. def, c-use, p-use). By analysing the syntax tree and symbol information, the flow graph also has *def-use* path information. The flow information such as block, decision, c-use, and p-use is stored at a separate file and the file is used by runtime library to analysis test coverage.

Instrumentation inserts a *probe node*, indicating the instrumentation pointer at source code, for a block in the syntax tree. *cSuds* uses probe routine and comma operator to instrument C code, but Ada 95 provides only probe routine to handle several cases. For example, in *if statement* such as **IF** (*expression*) **THEN**, instrumentation inserts a probe routine and *and then* operator like **IF** (*probe()*) **AND THEN** (*expression*) **THEN**, where the probe routine always returns true value. In case of iteration and selection statements such as *for*, *while*, and *case*, instrumentation inserts probe code before and after the statement to correctly reflect the behavior. Initialization statement in declaration part may have control transfer constructs such as procedure call. In this case, instrumentation adds a dummy symbol in addition to a probe routine. In exceptional handling, it may be impossible to know the source of control transfer in static time, so user must identify dynamic behavior after execution. In general, the probe is inserted at the beginning of a basic block.

Code generator generates an instrumented Ada code based on the syntax tree. During the traversal of syntax tree, it generates original Ada source code for regular syntax tree node. But it generates instrument statement for probe node that will call runtime-library during the execution of instrumented Ada code.

3.3. Runtime library

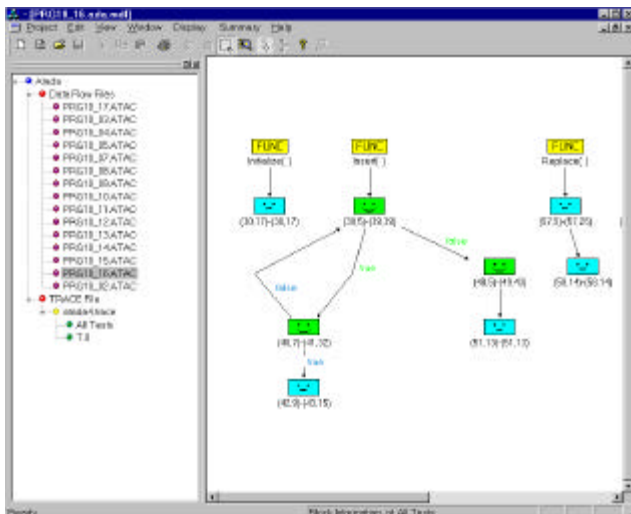
To collect execution traces, a set of Ada libraries is built in separate packages. The runtime library is a set of data structures, procedures and functions. For each module that is instrumented, ADA-PINE will generate data-structures that indicate the basic information of the module. Instrumented Ada source code, data-structures and runtime library will be compiled and linked to be an executable unit. During the execution of this unit, the instrumented statements will call the functions and procedures of runtime library. The basic idea is, at the

beginning of each procedure, an initialization will be done by insert the data structure of this unit into a link of data structures in the memory. After this, each instrument statement's execution will indicate the execution of the basic block that present. So the runtime library will record the times of each execution in one of the data structures and print out the first execution of each basic block into a trace file. Just about to finish the executing of the unit, the runtime library will dump the link of data structure and print the runtime of each basic block into the trace file.

How to detect the exit of Ada program is a difficult part of implementation because Ada does not have function call to detect the exit of all functions. For Ada95, we can implement this function by using controlled types of Ada language. When the instrumented code quit, an object executes a finalization before destruction. By this, we can detect the exit of Ada program, dump the link of data-structure in finalization. For Ada-83 user, this trick is useless because Ada-83 does not have object oriented programming feature. The only solution we found out is to add a super 'main' function that calls dump function after calls original 'main' function of instrumented code.

For Ada has concurrent feature, we must consider task identification and execution environment during implementation. We know every task instance of same type share same code section. By using Ada's task id feature, we make a task-table that includes task instance name and ids. When a new task instance is generated, the name of this instance and its id will be added into task-table. By using this table, we know which task instance is running the shared code. For same reason, each Ada task instance should have their own data-structure to record the execution time.

Considered semantics of task dependence and termination, we build context when each thread of control is established. When a thread of control finished execution, it will not be deleted until all thread of control depend on it finished exit.



3.3. Coverage analyzer

The coverage analyzer reads the source code, the flow information, and the test execution trace information. The analyzer displays a source program which marks covered or uncovered position with test results generated through each test case for block coverage and decision coverage. The test coverage displayed through control flow graph provides testers with more visualized view in order to help find appropriate test input. The test statistics is supported with a printable text report. Figure 4 shows control flow graph and test coverage in coverage analyzer.

4. Conclusions

In this paper, we showed a tool to measure the test coverage based on several coverage criteria. These criteria are block coverage, decision coverage, c-use coverage, and p-use coverage.

ADA-PINE is now a working prototype and it can be integrated with χ Suds to perform software test analyses. The flow graphs and the trace files generated by ADA-PINE are compatible with χ Suds format. The trace files currently contain block and decision coverage information. The ADA-PINE instrumentation can be done to all or selected program packages and is independent of Ada compiler. The analysis for Ada tasking, generic, and subprograms overloading has not been implemented at this stage.

5. References

- [1] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers: Principle, Techniques, and Tools," Addison Wesley, Readings, MA. 1986.
- [2] J. G. P. Barnes, "Programming in Ada 95," Addison-Wesley, Readings, MA. 1995.
- [3] " χ Suds User's Manual," Telcordia Technologies, Inc. 1998.
- [4] P. G. Frankl and E. J. Weyuker, "An Applicable Family of Data Flow Testing Criteria," *IEEE Trans. on Software Engineering*, 14(10):1483-1498, October 1988.
- [5] J.R. Horgan and S.A. London, "Data Flow Coverage and the C language", in *Proceedings of the Fourth Symposium on Testing, Analysis, and Verification*, pp. 87-97, Victoria, B.C., Canada, October 1991.
- [6] Intermetrics, Inc.; *Ada Reference Manual - Language and Standard Libraries*, ISO/IEC 8652:1995.
- [7] J. R. Levine, T. Mason, and D. Brown, "Lex & Yacc," O'Reilly & Associates, Inc., 1992.
- [8] S. Rapps and E. J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Trans. on Software Engineering*, SE-11(4):367-375, April 1985.

[9] H. Zhu, P. A.V. Hall, and J. H.R. May, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys*, 29(4):366-427, December 1997.