OCA contact: John B. Schonk
894-4820

Sponsor technical contact

PHILIP W. WATTS
(615)574-5142
MARTIN MARIETTA ENERGY SYSTEMS
P.O. BOX M
OAK RIDGE, TN 37831

Sponsor issuing office

FREDA HOPPER
(615)576-0251
SAME

Security class (U,C,S,TS) : U
ONR resident rep. is ACO (Y/N): N

Defense priority rating :
GOVT supplemental sheet

Equipment title vests with: Sponsor X
GIT

Administrative comments -
PROJECT INITIATION
GEORGIA INSTITUTE OF TECHNOLOGY
OFFICE OF CONTRACT ADMINISTRATION

NOTICE OF PROJECT CLOSEOUT

Closeout Notice Date 02/01/90
Original Closeout Started ********

Project No. B-10-646 Center No. R6415-0A0
Project Director McCracken W M School/Lab OIP
Sponsor Oak Ridge Nat'l Lab/Martin Marietta
Contract/Grant No. 19K-CN982C Contract Entity GTRC
Prime Contract No. DE-AC05-840R21400
Title Verif. & Valid. of Functional and Non-Func. Req. for Reusable ADA Modules
Effective Completion Date 890930 (Performance) 890930 (Reports)

<table>
<thead>
<tr>
<th>Closeout Actions Required</th>
<th>Y/N</th>
<th>Submitted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Final Invoice or Copy of Final Invoice</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Final Report of Inventions and/or Subcontracts</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Government Property Inventory &amp; Related Certificate</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Classified Material Certificate</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Release and Assignment</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

Subproject Under Main Project No. ____________________________
Continues Project No. ____________________________

Distribution Required:

<table>
<thead>
<tr>
<th>Distribution Required</th>
<th>Project Director</th>
<th>Administrative Network Representative</th>
<th>GTRI Accounting/Grants and Contracts</th>
<th>Procurement/Supply Services</th>
<th>Research Property Management</th>
<th>Research Security Services</th>
<th>GTRC</th>
<th>Project File</th>
<th>OCA/CSD</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Director</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Administrative Network Representative</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GTRI Accounting/Grants and Contracts</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Procurement/Supply Services</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Research Property Management</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Research Security Services</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GTRC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project File</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCA/CSD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: Final Questionnaire sent to PDPI.
First In Process Review

Verification and Validation of Reusable Ada Software

1 December 1987

Supported By

Martin Marietta Energy Systems, Inc.

Software Engineering Research Center
Georgia Institute of Technology
Atlanta, GA. 30332-0280
1. Progress

During this period we have initiated Subtask 1. The activities of the first subtask have been to review the literature for recent developments in V&V that may be applicable to our project. We have completed that review and as we suspected no new research has been conducted in this area.

Our initial formulation of a technique to aid in performing the V&V of reusable Ada is to use error based testing techniques as the foundation from which to expand to cover the requirements of the project. We have selected error based technologies because of their ability to quantify confidence in test results, as typically manifested in mutation test scores (mutation is an error based technique). Error based testing is founded on the principle that test cases can be developed that reveal the presence or absence of specific types of errors. Therefore, you can develop a confidence (numerically) that your test cases have adequately tested for those specific types of errors. References to mutation and error based testing include the text by DeMillo, McCracken, et al, "Software Testing", Benjamin Cummings Publishers.

We have also initiated discussions concerning the requirements for V&V of Ada Reusable Components as outlined in subtask 2. Our early concerns are with comprehending the differences between the somewhat easier task of testing components specifically developed for reuse from the more difficult task of applying those components in a reuse application. In addition, the task of using already developed components in a reuse application has to be addressed. Our early intuition leads us to feel that error based techniques will be the keystone of the solution.

During this period we have met with Wil Tracz, an IBM Fellow, who is currently at Stanford and is working in the areas of Ada and Reusability. He feels that our approaches are plausible and has given us some contacts at the Toshiba Corporation. Toshiba is doing some pioneering work in reusability as a part of the Japanese Fifth Generation Project.

2. Plans

During the next period we plan on developing further our error based approach to V&V of reusable Ada. This will require us to develop a methodology to understand the relationship of test results on a tested item to the expected results when it is placed in a new application. In addition, we will look at the applicability and, if necessary, the modifications to the error based approach for the V&V of non-functional requirements. We will also need to develop the error types associated with Ada and reusing Ada. The development of error types will have to be performed without the benefit of error studies, since to our knowledge none of this data is available. One of the tasks that we may suggest for future efforts is to collect or analyse collected data that will allow us to use real error data to develop the error types associated with Ada and reuse of Ada. We will recheck the field to insure that no
V&V OF REUSABLE ADA SOFTWARE

- CURRENT STATUS

- PLANS
CURRENT STATUS

- SOME ISSUES AND QUESTIONS

- ARE COMPONENTS DEVELOPED FOR REUSE OR DEVELOPED AND REUSED? (i.e., IN A SYSTEM AS WELL AS AN INDIVIDUAL CONTEXT)

- ARE WE ONLY CONCERNED WITH VALIDATION OR IS VERIFICATION AN ISSUE AS WELL?

- HOW DOES REUSE AFFECT THE V&V OF NON-FUNCTIONAL REQUIREMENTS?

- ISN'T REUSE A PORTABILITY PROBLEM?

  - HARDWARE ENVIRONMENT PORTABILITY —
    CAN I DEVELOP TRULY HARDWARE INDEPENDENT SOFTWARE?

  - SOFTWARE ENVIRONMENT PORTABILITY —
    CAN I DEVELOP SOFTWARE THAT WILL RUN CORRECTLY IN ANY USAGE CONTEXT?
CURRENT STATUS (CONT.)

- A SIMPLER APPROACH
- DEVELOP COMPONENTS FOR REUSE (e.g., EVB'S GRACE)

- THE PROBLEMS OF V&V FOR THIS APPROACH
- TESTING FOR CORRECTNESS
- HOW DOES THE COMPONENT BEHAVE IN AN APPLICATION?

- SOME APPROACHES THAT MAY BE TAKEN
  - "PARAMETERIZATION" — CONSTRAIN THE INPUT SPACE TO A TESTABLE SET
  - CONSTRAIN NON-FUNCTIONAL REQUIREMENTS — ON MACHINE X, COMPONENT WILL TAKE Z UNITS, UNDER SUCH AND SUCH CONDITIONS
  - ABSTRACT NON-FUNCTIONAL REQUIREMENTS — USE ALGORITHM ANALYSIS TECHNIQUES AS A SELECTION GUIDE (e.g., BOOCH)

- OBVIOUS DETRACTORS
  - "PARAMETERIZATION" LIMITS APPLICATION DOMAIN WITHOUT MODIFICATION (BUT HELPS PORTABILITY IN SOFTWARE ENVIRONMENT)
  - HARDWARE ENVIRONMENT DESCRIPTIONS ARE USUALLY LIMITED
  - ALGORITHM ANALYSIS TECHNIQUES ARE USEFUL ONLY AS GUIDES
CURRENT STATUS (CONT.)

- ANOTHER APPROACH
  - DEVELOP A COMPONENT IN A SYSTEM, BUT PLAN ON REUSING IT OR DECIDE TO REUSE IT.
  - THE SAME ISSUES OCCUR (BUT THEY ARE MORE COMPLEX)
    - HARDWARE ENVIRONMENT PORTABILITY
    - SOFTWARE ENVIRONMENT PORTABILITY
CURRENT STATUS (CONT.)

- PROPOSED APPROACH
- DEVELOP CONFIDENCE MEASURES BASED ON ERROR BASED TESTING TECHNIQUES

"THE GOAL OF ERROR BASED TESTING IS TO CONSTRUCT TEST CASES THAT REVEAL THE PRESENCE OR ABSENCE OF SPECIFIC ERRORS."

\[ B(T) = x \]

CONFIDENCE OF \( B(T) = x \) IS THE SAME REGARDLESS OF THE APPLICATION
BUT!!!

(REMEMBER, CONFIDENCE IS THE CONFIDENCE I HAVE THAT GIVEN A SET OF TEST DATA T, I HAVE A MEASURE OF ITS ABILITY TO REVEAL THE PRESENCE OR ABSENCE OF SPECIFIC TYPES OF ERRORS)

CONFIDENCE OF \( H(T_a) \) N.E. \( H(T_{a'}) \)

I.e., if you test B in an environment P with A as a source of inputs and you move B to a new application with A' as a source, you no longer know the confidence of B relative to the new environment.
PLANS

- Use confidence measures to quantify components "testedness" in some domain – 

- Understand the relationship of $\Phi$ to a new application domain $\Phi'$

By

- Developing a methodology that helps us understand $\Phi$, $\Phi'$

- Understand ADA and reuse error types

- Expand or modify error based techniques to include non-functional requirements
Second In Process Review

Verification and Validation of Reusable Ada Software

9 March 1988

Supported By

Martin Marietta Energy Systems, Inc.

Software Engineering Research Center

Georgia Institute of Technology

Atlanta, GA. 30332-0280
1. PROGRESS

During this reporting period we have initiated and completed some parts of subtask 2. Included in this task are two elements, methodological and experimental. We are using the methodological aspect as a top down view of the V&V problem and we are using the experimental aspect as a bottom up view. These two attacks will produce a set of guidelines which are substantiated by experimental evidence that gives credence to them. Our report this month includes a writeup of a description of our work and preliminary results. One specific aspect of our work that deserves recognition is the development of our experiment harness. Since our approach requires a method of producing error based testing experiments, we needed to create an environment that would allow us to conduct them. We have successfully created that environment and have conducted preliminary experiments to validate it as well as confirming some of our early conjectures.

1.1. Introduction

The purpose of this research is to investigate the verification and validation (V&V) of reusable software written in the Ada programming environment. Reusability is attractive because it can potentially reduce the amount of effort required to develop software. But reuse has its costs, as Booch has pointed out.

The cost of the generality in producing a reusable component is greater than the cost of developing a less general solution. The component will be more severely stressed in use, and will therefore be likely to have greater long run maintenance costs. Also, because the component is being reused, there must be a configuration management system available so that when a change is made, its instantiations can be updated. Matching and retrieval (and understanding) of components also implies a cost. There may also be managerial, legal, and social issues that imply costs.

As far as V&V is concerned, the important point is made in the second sentence. Reusable components are more general and intended to be used in a wide variety of situations. The V&V of reusable components is responsible for assuring this.

In traditional software development, an "error" is an instance where the behavior of the software differs from its specification. The first question that arises is, therefore, does the definition of an error change when reusable software is considered? We feel intuitively that the answer is yes, and we recognize that we need to justify this feeling. To develop this idea, we first discuss what we mean by reuse.

We then present two approaches we are making to understanding how the V&V of reusable components differs from that of traditional software.

1.2. Reusability

Reusability has been a major concern of software developers for some time. Books cover the topic, typically offering lists of "portability concerns" of which the developer should be aware. Tools exist to check for non-portable constructs (e.g., Lint in the case of C and PFORT in the case of Fortran). In these situations, a portability error is defined to be a program construct that violates the list in the book or that is detected by the tool.

If an attempt is made to specify the portability of a program, the specification is normally made by listing the machines/operating systems on which the program should run. The implication to V&V is that the problem is multiplicative--all features have to be tested in all environments. It is
attractive to try and characterize portability errors in such a way that the problem is easier to deal with. What seems necessary is an extension of the concept of "error" to include situations where unspecified assumptions of the program's operating environment are made explicit.

It is also possible to reuse a component by incorporating it into a new application environment. In the past, a library function or collection of functions is accessed using traditional linkage mechanisms. There is little or no customization of the component for the new situation. Of course this is not a black or white decision. Library functions are sometimes modified before being applied to a new situation. But the cost of understanding someone else's source code is high. Moreover, V&V considerations are often vitiated.

Ada offers another possibility. The generic facility allows developers to "instantiate" the component in a variety of ways, depending on the requirements of a new application. The characteristics that may be altered are delineated in such a way that the developer need only look at the specification and not the body of a generic component in order to understand what it does.

We would like to understand just how reusable a generic is. To do this we would like to exercise (exorcise) implicit constraints. Once again, these are not bugs. In this case as well as in the case of portability, we are generalizing the specification of the problem to include considerations beyond those intended when the original design/implementation was performed. For example, does a generic assume that a function parameter is be commutative? Does an arithmetic operation rely on a certain minimum range for integer operands?

1.3. Approaches

We have undertaken two approaches to understanding the V&V of reusable software. The first is methodological. It is important to understand how reusable software is different from traditional software. For example, if we are confident that we have adequately tested a component in one situation, what can we say about how well it will work in another. Is the test data that we used applicable to the new situation? How relevant is the programming language used? Which V&V methodologies are most applicable?

The other approach is experimental. What can we learn from trying to actually test some components that were written to be reusable? What impact does the programming language have on the testing task? How appropriate is a particular methodology to the V&V of reusable components?

The methodological approach is analytic and top down. It attempts to understand the V&V of reusable components by looking at the total life cycle and understanding how reuse might perturb traditional methodologies. The experimental approach is bottom up and synthetic. We need more information about the V&V of reusable components to justify our analysis. To obtain this data we have built a prototype Ada Mutation system capable of dealing with operand mutations. It allows us to run experiments to learn more about specific Ada constructs such as generics. The next two sections go into more detail about the two approaches.

1.4. Some Methodology Issues

A V&V methodology for reusable software must concern itself with the entire life cycle of software development. Thus far, we have been concerned with asking questions and raising issues. This has been accomplished by brainstorming sessions and by a review of the literature. The following issues appear promising.

Will software components that are intended for reuse use the Ada language in ways that are different from those not so intended? Are there misuses of the Ada language when used to build reusable components that can be detected?

If a reusable component needs to be tested more than a traditional component does that imply that the amount of integration testing of a system that uses these components can be reduced? Is there a trade off between unit testing and integration testing?

Generic units are parameterized by types and subprogram units. Each set of supplied parameters defines a new instantiation of the unit. Thus, generic units can be tested in a non-traditional
fashion, by instantiating it in various ways and then testing the instantiations. Is there a trade off between the breadth of the instantiations tested and the degree to which (depth) each instantiation is tested? This may characterize the differences between the White Box and Black Box testing of reusable Ada components.

Design methodologies have been developed for components intended for reuse. To the extent that these differ from traditional techniques, are there associated alterations required to V&V methodologies?

Are there metrics of reuse? Is it possible to measure the portability or adaptability of a component? What we intend to look at here is the idea of extending the "adequacy" score from Mutation Analysis to the domain of reusable components.

If a reusable component is tested in a different way than a traditional piece of software, must the tests themselves be managed in new ways? What implications are there on testing harnesses if the software is intended for more general use?

Booch has proposed certain ways in which an abstract software component could be generalized into different versions that might prove useful in different situations. For example, an abstract data type for a stack might have versions with a fixed upper limit in size and one that can grow as large as machine memory permits. Other options allow for garbage collection versus language-provided storage management. If he has successfully captured "axes" of generality that hold across a large class of components, then it may be possible to construct a test harness that exercises components in these areas. For example, one test may keep requesting new instances of a data structure until memory is exhausted. Another may obtain and release objects to test for storage fragmentation problems. These tests would be largely independent of the particular data type being used.

1.5. The Experimental Approach

To more fully understand these issues, we have undertaken to study the appropriateness of the Mutation Analysis to the V&V of Ada programs. Ada is interesting both in the areas of portability and adaptability. Ada was designed to be a portable language, and features have been added to make explicit machine differences. It may be possible for Mutation Analysis to make use of these features and so derive indications of where a particular construct may not be portable.

In the area of adaptability, the Ada generic capability is particularly well suited to the development of parameterized components that can adapted to a variety of situations. Mutation Analysis can be used to obtain an indication of the adequacy of a set of test data in exercising a software module. We would like to understand how these two ideas interact. Are there particular Ada features that are difficult to deal with using Mutation Analysis? How do the adequacy scores relate to the reusability of a component?

The experimental approach attempts to improve our understanding of the V&V of reusable components by applying a specific methodology to a specific programming language. We hope to learn of issues that specifically relate to the testing of reusable components. There are two aspects of this approach. On the one hand, we need to know whether the specific tools that have been developed can be adapted to deal with Ada constructs. To investigate this, we have adapted an Ada parser to generate information about Ada constructs that are candidates for mutation. The information is placed into a file that can is later processed to produce mutated Ada programs. The mutated programs are then run through an Ada compiler. Programs that successfully compile are exercised with a set of test data. Mutants are killed and equivalenced in the same manner as with traditional mutation systems.

The other aspect of the experimental approach is to look at the Ada language directly and to try and understand issues related to mutation testing. What can we learn from mutating an uninstantiated generic unit? Are there tests that can be made that indicate portability problems?

Thus far, we have succeeded in building a prototype Ada mutation system. It is capable of dealing with the mutation of Ada operands. We chose this class of mutation because it requires significant changes from how it is handled in Fortran. Ada is a block structured language with complex visibility rules. Fortran strictly limits the visibility of all operands.
We have tried some simple experiments with the prototype system. We took a small generic unit in its uninstantiated form and mutated it. The particular unit applies a user-supplied function to three arguments. It relies on the fact that the function is associative and commutative. When normal mutation operators are applied, all mutants are killed. By adding new mutation operators for commutativity and associativity several of the new mutants are not killed, indicating that an assumption has been overlooked in the testing.

The experiment indicates two things. First that it is feasible to use mutation analysis on Ada programs, at least in the case of operand mutation. Second, that mutation analysis can lead us to discover aspects of reusability that need to be treated in other than a traditional way. We expect to be able to use the prototype system to experiment with portability reuse as well.

1.6. Conclusions

When Parnas attempted to characterize how software should be partitioned into modules, he emphasized that the number of assumptions that modules make about each other should be minimized. His main concern was to facilitate the maintenance of systems composed of such modules when they are adapted to another context or enhanced with a new purpose. He specifically extended the notion of assumption beyond the passing of arguments across a procedural interface. The detection and limitation of these assumptions is a problem that is still with us and that we hope to address in this research.

2. Plans

We expect to explore two areas during the remainder of the contract. First, we would like to better characterize what a reuse error is and how reuse errors differ from traditional software errors. Second, we would like to learn what it takes to detect such errors; in particular, can traditional V&V methods be adapted.
Appendix A

Presentation Materials
V & V OF REUSABLE ADA SOFTWARE

IN PROCESS REVIEW
NUMBER 2
9 MARCH 1988

Software Engineering Research Center
Georgia Institute of Technology
Atlanta, Georgia

Contract No. 19K-CM982C
LONG TERM GOAL

- Development (or Modification of Extant) Methodologies for the Verification and Validation of the Functional and Non-Functional Requirements of Reusable Ada Components
WHAT ARE THE KEY CONCERNS THAT AFFECT THE V & V OF REUSABLE COMPONENTS?

1) PORTABILITY (previously defined as hardware environment portability)

- The effect of moving components across hardware and operating system environments
WHAT ARE THE KEY CONCERNS THAT AFFECT THE V & V OF REUSABLE COMPONENTS?

2) ADAPTABILITY (previously referred to as software environment portability)

- The effect of moving components across different applications
- A further complication is the instantiation of generics in different environments!
RESEARCH APPROACH

- Based on intuition -- we feel that there exists a set of errors we will call reuse errors that are not totally contained in the set of "correctness errors."
RESEARCH APPROACH

• "Correctness Errors" -- Those errors where $P(T) \neq F(T)$ (Program does not satisfy specifications of its intended behavior for some element of T).

• These are sometimes referred to as failures to write specifications that correctly represent the design intent.
RESEARCH APPROACH

Given that there exists a set of reuse errors that may not be contained in the set of "correctness errors" - How do we determine them and how do we minimize them?
RESEARCH APPROACH

2 APPROACHES

APPROACH A - Examine the life cycle and determine what can be done to minimize all errors, with particular attention paid to reuse situations

APPROACH B - Develop an experimental vehicle to confirm or refute conjectures of reuse errors and use these results to develop a class of reuse errors
RESEARCH APPROACH

Methodological Investigations:

Review the life cycle and determine what V & V steps need to be modified to accommodate reuse.

e.g., testing instantiations of generics may imply a breadth of testing vs. a depth of testing tradeoff.
RESEARCH APPROACH

Methodological Investigations:

**Breadth of Testing vs. Depth of Testing:**

**Breadth** - How many instantiations do I test?  
**Depth** - What is the degree to which each instantiation is tested?

This may have some relationship to black box vs. white box testing strategies.
RESEARCH APPROACH

EXPERIMENTAL INVESTIGATIONS

- Determine that we can develop a test harness for reuse experiments
  - Developed the Ada Prototype Mutation System

- Use the harness to conduct experiments
  - such as for portability concerns
RESEARCH APPROACH

EXPERIMENTAL INVESTIGATIONS

• For example, is there a class of detectable uninstantiated generic errors?
  - This implies test cost reductions

• Define operators capable of detecting associativity / commutivity dependencies, and thus detect a misapplication of a generic
METHODOLOGY ISSUES

- Ada language survey
  - portability problem detection

- Effect of reuse development methodology on V & V methodologies
  - reusability guidebook
  - reuse V & V's feedback to specification technology
METHODOLOGY ISSUES

• Testing of reusable components
  – e.g. testing an uninstantiated generic
    versus generating many instances
    and testing them
  – New possibilities for manipulation
    types
    functions
Methodology Issues

- Extensions to the idea of what is a test
  - What can we conclude after testing a reusable component when we make use of it?
  - Adequacy scores
Ada Prototype Mutation System

Ada Source Code File → Mutant Generator Pipeline → Mutant Source Code Files

Mutant Descriptor Record File → Ada Compiler → Executable Object Code

User → Test Case Generator → Test Cases → Executor

Expected Output

UNIX

VMS
Ada Prototype Mutation System
Mutant Generator Pipeline

1. Ada Source Code File
2. Source Code File (With Harness)
3. Source Code Parser
4. Symbol and Reference Table
5. Source Code Line Generator
6. Mutant Descriptor Records
7. Filter
8. Mutant Descriptor Record File (No Duplicates)
9. Mutant Source Program Generator
10. Mutant Source Code Programs

Hand Edit
EXPERIMENT: TESTING OF GENERICS

GENERIC
    type Gen_Type is private;
    with function "+" (a, b: in Gen_Type)
         return Gen_Type;
...

package body Example_Package is
    function op_of (a, b, c: in Gen_Type)
         return Gen_Type is
        begin
            return c + b + a;
        end;

end Example_Package;
EXPERIMENT: TESTING OF GENERICS

<table>
<thead>
<tr>
<th>TEST CASE 1</th>
<th>TEST CASE 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gen_type =&gt; int.</td>
<td>Gen_type =&gt; int.</td>
</tr>
<tr>
<td>&quot;+&quot; =&gt; &quot;+&quot;</td>
<td>&quot;+&quot; =&gt; &quot;-&quot;</td>
</tr>
<tr>
<td>a =&gt; 10</td>
<td>a =&gt; 10</td>
</tr>
<tr>
<td>b =&gt; 5</td>
<td>b =&gt; 5</td>
</tr>
<tr>
<td>c =&gt; 1</td>
<td>c =&gt; 1</td>
</tr>
</tbody>
</table>

NEW MUTANT TYPE: COMMUTATIVITY OPERATOR

replace \( c + b \) with \( b + c \)
replace \( c + b + a \) with \( a + (c + b) \)

RESULTS

TEST CASE 1: All standard mutants killed
Two new mutants not killed

TEST CASE 2: Two new mutants killed
WHERE DO WE GO FROM HERE?

- Obtain a better characterization of reuse errors and learn how reuse errors differ from correctness errors.

- Learn how to detect reuse errors and how traditional V & V methodologies can be adapted to deal with them.
VERIFICATION AND VALIDATION OF
REUSABLE ADA* COMPONENTS

C. K. Bullard, D. S. Guindi, W. B. Ligon, W. M. McCracken,
S. Rugaber

Software Engineering Research Center
Georgia Institute of Technology
Atlanta, Georgia 30332

This work is supported under contract 19K-CN982C under Prime
Contract DEAC05-84OR21400 through Martin Marietta Energy
Systems.

THE PROBLEM

Introduction: Motivation for the Work

The purpose of this research is to investigate the verification and
validation (V&V) of reusable software components written in the Ada
programming language. Software reuse is the process of using a software
component in a context other than the one for which it was originally
developed. Reuse is attractive because it can reduce the amount of effort
required to develop software as well as improve the quality of the
resulting systems. Design and implementation costs can be reduced if pre-
existing components can be utilized, and V&V costs can be reduced if the
components have been adequately tested when they are built instead of
having to be completely reexamined every time they are used.

If a component is independent of environmental peculiarities, then V&V
efforts taken during development need not be redone when the component is
reused. Unfortunately, such independence is unlikely. Dependencies can
arise from the hardware, the operating system, the compiler, or the
application environment in which the component is used. A viable strategy
for the V&V of reusable components must address these dependencies and try
to move as much effort as possible to the development stage where it need
be performed only once. The purpose of this paper is to explore these
special requirements that reusable components place upon the V&V process
and then to propose methods and techniques to satisfy them.

One objective of the research is to determine the applicability of
techniques and methods from traditional V&V to the V&V of reusable
components. The next subsection presents definitions of terms from

* Ada is a registered trademark of the U. S. Government - Ada Joint
Program Office.
traditional V&V that are used in the remainder of the paper, after which concepts related specifically to the V&V of reusable components are introduced. The next major section examines several V&V techniques and their applicability to detecting reuse problems. The two sections that follow this examine reuse issues in detail and present characterizations of those issues. The first is concerned with portability problems, and the second is concerned with adaptability. Reusable components can be used in environments quite different from those imagined by the original designers, implementors and testers. The final section discusses how this freedom overwhelms traditional V&V approaches, and what can be done to address the problem.

**Traditional V&V**

Verification and validation were originally thought of as activities that occur at the completion of the development phase of the software lifecycle, whereas contemporary methods emphasize that V&V activities be carried out throughout the lifecycle (DeMillo et al., 1987). Thus V&V encompasses activities beyond those referred to as testing. Testing may be thought of as a proper subset of V&V, in that it does not normally include static methods, nor does it typically occur until after implementation, at the unit, integration and system levels.

V&V is a sub-process that occurs as a part of the overall software development process. V&V has been formally defined by the Joint Logistics Commanders (Joint Logistics Commanders, 1983) as: "Verification is the iterative process of determining whether the product of each step of the computer software development process fulfills all requirements levied by the previous step." "Validation is the evaluation, integration, and test activities carried out at the system level to ensure that the finally developed CSCIs [Computer Software Configuration Items] satisfy the user's and supporter's requirements set down as performance and design criteria in the system and software requirements specification." These definitions and common practice dictate that software be verified at every step of the software development process and that validation be conducted as well.

Many techniques are used to achieve the required levels of verification and validation. The remainder of this section defines the terminology typically used to describe these activities. The first four terms are used to describe methods of V&V by what they do and how they do it. That is, analysis methods are describing what types of V&V can be conducted (static or dynamic) and testing strategies describe how the V&V is conducted (white or black box). Following those terms is a description of unit and integration testing. The goal of any V&V method (strategy or technique) is to minimize the errors in the deliverable software and to do that at a minimal cost. These definitions are taken from DeMillo et al. (1987).

**Static Analysis.** The verification of a software product by observing or analyzing the product without executing it. Static analysis includes but is not limited to structured walkthroughs, design reviews, desk checking, and checking for adherence to standards, as well as the automatic analysis of program properties such as the occurrence of uninitialized variables.

**Dynamic Analysis.** The verification of a software product by executing the software and observing its behavior in reference to its specification. Some examples of dynamic analysis are coverage methods such as path analysis, domain testing and mutation analysis.

**White Box Testing.** Includes testing techniques that determine
correctness by observing the internal structure and logic of the program under test.

**Black Box Testing.** Includes testing techniques that determine correctness by observing the functional (i.e., input/output) behavior of the program under test.

**Unit Testing.** Testing of individually compilable units of software such as subprograms, procedures, routines, etc. These tests typically rely on the aforementioned analysis methods and testing strategies to achieve the required degree of confidence. This level of testing is concerned with whether or not the unit meets the specification of its intended behavior. This is sometimes referred to as testing for correctness.

**Integration Testing.** Integration testing is a broad category that usually includes sub-system as well as system testing. It is concerned with the correctness of interfaces between units as well as validation of a system's function.

To minimize the cost of V&V, it is desirable to reduce the amount of integration testing while possibly increasing the amount of unit testing. This is based on the observations that unit testing is less expensive to conduct and that the cost of correcting errors rises with the increasing level of integration of the software. In addition, testing techniques tend to be more effective at the unit level whether they be formal verification or mutation testing.

**Formal Methods.** In addition to testing and static analysis, formal methods are sometimes used to demonstrate the correctness of a program. Formal methods, sometimes called formal verification or program verification, are concerned with formally proving that a program is correct with respect to its specification, or in some cases proving the correctness of the design of a system. In addition to formal verification, formal techniques are sometimes used to specify a program without the use of rigorous proof techniques.

**Specification.** The description of the intended behavior of a program. Specifications range from formal, based on mathematical logic, to completely informal using natural language descriptions of the functions the program is to perform.

**Adequacy.** (Budd and Angluin, 1982) A term used to describe how well a set of test data exposes errors in a program based on a specific criterion. An example of this is how well a set of test data exposes mutation analysis error classes. An adequacy criterion can be used to indicate when a testing method may be terminated.

**Requirements.** The properties and constraints that a software system must satisfy. Included in the requirements are the non-functional and functional requirements of the system.

**Functional Requirements.** The services or features of the software system required by the user; its behavior.

**Non-functional Requirements.** The constraints or restrictions placed on the software system, such as reliability, timing and sizing limitations, and maintainability.

**Mutation analysis.** (DeMillo, 1988) A test technique that measures the error exposing ability of test data on a program by creating slightly
changed versions of the program and seeing how the test data detects those changes.

V&V for Reuse

Reuse Errors. Normally, in developing software, one of the major difficulties that arises is software errors. A software error is an anomaly that causes a component to behave differently from its specification. In software reuse, it may be the case that a component's behavior meets its specification in its intended environment, but fails to meet its specification in some other environment. For example, if a component depends on the order in which a compiler evaluates the subexpressions of an expression, it may not behave as intended when compiled with a compiler employing a different policy. This type of anomaly is called a reuse error.

This definition of a reuse error raises an interesting question: if a component fails to meet its specification in some environment, is the component in error, or should the specification have required the component to operate properly in both the original and the new environments? Currently, issues of environment compatibility are considered non-functional requirements, if they are mentioned at all: "The system will be targeted for an XYZ-2000 processor under OS with the X compiler." The advent of software reuse, however, demands that environment compatibility be considered as an explicit requirement, if the goal is to produce software that is independent of its environment, and therefore as reusable as possible. This question may well be answered by the method used to "repair" the error: if the code changed, it is a coding or design error, if the specification is modified, it is a specification error.

Portability and Adaptability. In attempting to classify and understand reuse errors, there are two different kinds of environments to consider: a machine environment and an application environment. The machine environment includes not just the physical hardware, but a virtual machine that incorporates the operating system and any run-time environment and conventions set up by the compiler. In a sense, the machine environment includes all that the reusable software component regards as "the system" and is unchangeable (system modifications notwithstanding). The application environment, on the other hand, is composed of other application-programmer written components with which the reusable software component is to be used. The application environment is essentially the "user" of the reusable component, and that use is generally changeable, since it is programmer code as opposed to system code.

Software reuse can therefore be divided into two activities: porting and adapting. To port a component is to change its machine environment, and to adapt a component is to change its application environment. Reuse errors are therefore divided into portability errors and adaptability errors. A portability error is one that occurs when a reusable component is moved from one machine environment to another, and an adaptability error is one that occurs when a reusable component is used in an application environment different than that for which it was originally designed. Effectively, portability and adaptability define two interfaces a reusable component must consider: an interface with the machine, or machine interface; and an interface with the application, or application interface. It is with respect to these two interfaces that reuse errors occur.

Characterization. The distinction between portability and adaptability is central to the understanding of reuse errors and affects the V&V methods used in the development of reusable software components. The major distinctions are as follows:
1) Portability errors are contained entirely within the reuseable component. Problems such as this can be detected during unit testing, as opposed to system testing.

Adaptability errors span the application interface and in fact represent only potential errors in the component itself. It is meaningless to say that an actual error exists until some sort of environment is present. The major difficulty is that it is impossible to anticipate all of the potential application environments in which the reuseable component will be used. These problems must be considered at system integration time, although the potential for an adaptability error may be detectable by considering the isolated component.

2) According to the Ada language definition, components that have portability errors are considered "erroneous". The Ada Language Reference Manual (Department of Defense, 1980) points out the places in the language where behavior is undefined or some latitude is given to the compiler writer or system. While such a component may behave correctly in one machine environment, its correctness is nonetheless incomplete. Such problems can often be detected with static analysis, as the occurrence of language constructs is easily recognizable.

A component with an adaptability error, on the other hand, may be perfectly correct in and of itself, but its correctness depends on the way in which it is used. Obviously, such a problem does not lend itself well to formal verification techniques since the "error" may not exist at all until the component is placed in a new application environment.

3) Portability has been studied for some time, and is relatively well understood. In addition, a considerable amount of research has been performed developing tools to detect portability errors. The bibliography lists several references to specific portability problems with Ada (Digital Equipment, 1985; Barnes, 1984; Nilssen, 1984), as well as several for other languages such as C (Johnson, 1979) and FORTRAN (Ryder, 1974). This does not imply that these portability issues have been solved, but Ada has managed to isolate many of them in such a way as to improve their manageability.

Adaptability, on the other hand, has traditionally been the bane of software testing and evaluation. Combining separately tested components into a system yields a combinatoric explosion of interactions that can yield errors. Most V&V techniques either become infeasible when used on an entire system or fail to address the interface problem and so suffer from a lack of effectiveness. The central problem in V&V for reusable software components is the integration/interface problem.

An Example. As an example, consider the Ada code fragment in figure 1. The function foo modifies the exported global variable A. Later, in procedure bar, a statement contains both A and a call to foo in the same expression. Since Ada does not define the order of evaluation of terms in an expression, this code contains a potential reuse error.

If the subprograms foo and bar and variable A are all part of the same reusable component, then this component has a portability error because the component will not behave correctly in a machine environment that uses an order of evaluation different from that expected by the programmer. A V&V method for reusable Ada components should detect this reuse error during unit testing.
A : Integer;

FUNCTION foo (parm : in Integer) RETURN Integer IS
BEGIN
  A := parm;
  RETURN A;
END foo;

PROCEDURE bar (parml, parm2 : IN OUT Integer) IS
BEGIN
  parml := A * foo(parm2);
END bar;

Figure 1. An example.

If, on the other hand, function foo and variable A are in the same reusable component, but procedure bar is in the application environment, then this component has an adaptability error because the component is dependent on its use by the application environment. In this instance the application environment's use has the potential for an order of evaluation error. This cannot have been detected by examining the component in isolation, but must be looked for during integration testing.

Many of the same issues and problems come up in both portability and adaptability. In many cases, adaptability errors are really portability errors that have been "split" such that part of the error is in the reusable component, and part is in the application environment. This comes from the rather interesting fact that many of these problems involve more than one point in the program. Note that in the example, it is difficult to say whether the implementation of function foo, or the statement in procedure bar is in error. This type of error is called a two point error.

Other Definitions. Other definitions of the terms portability and adaptability exist. A typical definition of portability is "a qualitative judgement on how easy it is to change a given program such that it can be recompiled by a different compiler and then run with the required behavior, generally on a different target" (Nilssen, 1984). Therefore, a completely portable program would not require any changes to compile and run in a new machine environment. This definition is consistent with the one above, except that in reuse V&V, a quantitative rather than qualitative analysis is desirable.

Popular definitions of adaptability are not so uniform. Wallis (1982), in Portable Programming, defines adaptable: "used to describe software that requires a significant amount of work to move to another machine ... simply by making changes that are regular enough to be made by computer". This definition describes adaptability as a special case of portability. Our definition of adaptability is similar to portability but involves changing the application environment instead of the machine environment. Lecarme and Gart (1986), in Software Portability, define adaptability: "the ease with which [a program's] properties can be
modified. Adaptation is made without changing the program environment, and remains adaptation only so long as the new properties are closely related to the old ones. These definitions differ significantly from the one above in that in both instances the adapted program is modified whereas the definition above is intended to deal only with cases where no component source code is actually modified, only the application environment. Under the definition above, pure software adaptation (with no porting involved) is accomplished by using a pre-compiled reusable component via the Ada \texttt{with} statement, and the only modification allowed is through changing the resources provided to the adapted component. If modified code is considered, all notions of pre-existing reliability are lost, because there is no way of knowing what changes have been made or how they affect the operation and functionality of the program.

DETECTION MECHANISMS

There are a variety of mechanisms for detecting errors in traditional V&V. One of the goals of this project is to study the exploitation of these techniques for detecting reuse errors. Reuse presents significant problems to overcome because analysis cannot always be performed at the component level, but must consider the interaction between components. At the same time, exhaustive testing of software component interfaces remains an intractable problem. Our approach is to understand the portability and adaptability issues at the component's machine and application interfaces. This understanding can guide the selection and combination of different V&V methods and techniques to comprise a solution that intelligently analyzes software component interfaces with minimal effort.

Towards this end, this section discusses some of the approaches that can be used to analyze or test for portability and adaptability errors. First, three traditional approaches are presented: simulation, static analysis, and mutation analysis. Finally, constraint-based analysis is introduced as a promising new technique for detecting reuse errors.

Simulation

The obvious first approach to testing for reuse errors is to simply test a reusable module in a variety of environments. This approach extends the traditional notion of testing to include input that describes the execution environment. The most noticeable disadvantage is that this requires a multitude of machines and compilers. The next problem is that it does not consider new machines and compilers that are yet to come to market. This problem can be somewhat lessened by constructing an interpretive system that is capable of simulating a wide variety of run-time systems with parameters to control machine properties such as word size, order of evaluation, memory initialization policy, and parameter passing conventions.

The same ideas apply to adaptability. A component can be tested via a testing harness that simulates a wide variety of applications environments. Again, the same problems are present as with the portability case, only they are more profound because the potential application environments in which a component might be used cannot be easily characterized by a set of well understood parameters.

The real problem with this approach is that there is no guarantee that simply running a component in each environment will, in fact, cause potential errors to occur. Also, this kind of testing produces no measure of adequacy, no way to gauge the correctness of the component, and no way of understanding what has and has not been considered in the event the
component is actually moved to a new environment.

On the other hand, simulation may be the only approach available to deal with portability problems that involve component interaction and timing issues such as those that arise with Ada tasks. In order to make this approach feasible, a testing method should include adequacy criteria that reduce the amount of testing needed and provide a measure of a test suite's effectiveness. As such, simulation is useful as a foundation for mutation analysis and constraint-based testing.

Static Analysis

The next approach is to look at static measures of a component's portability and adaptability. Techniques exist to statically analyze code and recognize program and language structures that are non-portable or non-adaptable. Such an approach has the advantage that it does not require a complex environment modeling system or a roomful of machines. Also, static analysis requires less overhead because the code is never actually executed. Moreover, it provides a definite measure of reusability. This is an appealing approach for detecting some portability errors because many machine dependencies occur as language constructs, are localized, and are well understood. An example of such a tool is the ADAMAT static analysis tool (ADAMAT, 1988).

Adaptability errors, on the other hand, pose a problem because dependencies are often in two parts. In a reusable component, there may be a potential reuse error, and the application environment may use the component in such a way as to realize the error. When a reuse error crosses the application interface in this manner, it is impossible to detect by analyzing the component in isolation. At best, a static analysis tool could detect those areas that create the potential for the error. In many of these situations, however, this potential cannot be easily removed. Therefore, since all of the target application environments are unknown, a static analysis tool has to be run on the entire application each time a reusable component is used.

For example, if the function foo and the variable A in figure 1 are part of a reusable component, a static analysis tool could detect that the function modifies an exported variable and is therefore not as adaptable as possible. It is not likely, however, that foo can be re-written in such a way as to remove this potential adaptability error. Using only static means, it is impossible to say whether foo will work properly with bar unless the combined application is analyzed.

The main disadvantage of static analysis is that in many cases it cannot do more than detect the possibility of reuse errors. Many reuse errors involve component interaction and timing issues, which are not exercised by static methods.

Mutation Analysis

Another approach is to use mutation analysis techniques to measure the adequacy of a set of test data with respect to a defined set of errors and reuse errors. Mutation analysis starts with a description of reuse errors and develops a set of mutation operators that create variant programs. A test data set's ability to distinguish the original program from the variants provides a measure of that test data set's adequacy in detecting the errors.

Previous research (DeMillo, Lipton, and Sayward, 1978; Budd, DeMillo, Lipton, and Sayward, 1978; Budd, Hess, and Sayward, 1980; Budd, 1980;
DeMillo and Spafford, 1986) has developed a set of mutation operators for traditional errors based on error studies (Goodenough and Gerhart, 1975). More recently, mutation operators have been developed for Ada (Bowser, 1987; Appelbe et al., 1988). Errors that involve component interfaces are handled in these systems by mutating the entire application. The question arises whether these operators are sufficient to detect reuse errors if the test data set is simulated in the various target environments. In other words, if test data set T is adequate to detect all standard mutants of program P in environment E1, and if P is executed on T in a new environment E2, then any of some class of reuse errors that exist in environment E2 will be detected.

Experimental evidence has shown, unfortunately, that this does not hold because it is possible to have more than one specific interface point between components (for example, by calling a function more than once), and it is possible to distinguish all of the generated mutants without exercising the path that contains the reuse error. To remedy this, the method could be altered to require that all of these paths be exercised, but then the number of mutants created, and their subsequent overhead, increases greatly. The number of paths that must be exercised needs to be limited to those that hold a potential for error.

An alternate approach is to extend mutation analysis by developing new mutation operators for reuse errors. This approach generalizes the notion of program input and output beyond simple values to include variables, types, subprograms, and runtime environment parameters such as order of evaluation policy, word size, and memory initialization policy. The tester would manipulate these inputs in order to detect the reuse errors introduced by a set of mutation operators. Obviously, this requires a machine simulation system such as that described above in order to work with portability errors.

Using this kind of extension, components can be tested individually for traditional errors and the application environment can be tested in conjunction with the component specifically for reuse errors. In this case, all paths need not be tested directly, only enough to test the critical elements of the component interface and its environment. Since the new operators focus on potential reuse errors that exist at the component interface, the number of mutants would be reduced.

The advantage of this approach is that there is a definite measure of the adequacy of the tests and the method is not dependent on running tests in all possible environments. In essence, by focusing on reuse errors, the various environmental parameters are considered independently and combination effects only need to be considered when they are necessary to detect a specific error. This approach also provides the possibility of working with timing-dependent problems, such as the Ada rendezvous, that cannot be achieved with static methods.

To investigate this approach, a prototype Ada mutation system was constructed and some experiments were run. One of the experiments involved adaptability testing for Ada generics that take a function as a parameter. A generic was written that depended on its formal functional parameter being commutative and associative. It was demonstrated that the current set of mutation operators could not detect this dependency. A new operator was defined that was capable of detecting this problem.

Unfortunately, the efforts so far have not been able to construct such an operator for many of the reuse errors. Instead, most of the proposed mutation operators result in mutants that are distinguished only if an error exists and input is provided that causes the erroneous behavior in
the original. If the original program is correct, the behavior of the mutant is always the same as the original. If the test data can distinguish the mutant, then there is an error, but if it cannot, there are two possibilities: either the program is correct or the test data is inadequate.

**Constraint Based Analysis**

Finally, there is the possibility of combining static analysis with mutation analysis and a relatively new technique - constraint-based analysis (Weyuker and Ostrand, 1980; Offutt, 1988). In this approach, a static analysis tool is used to detect and record potential reuse errors with a component's library information (figure 2). This record defines a constraint on the component's use that, if violated, could cause a reuse error. When the component is used in an application, the constraints are used to generate mutants that test the application environment to see if they in fact cause the error to occur (figure 3). For example, static analysis can detect that one of a reusable component's functions fails if one of its parameters is out of range. This information can be stored with the component's library module. The information can be used to generate a mutated program that raises an exception at each point the function is called if that parameter is out of range.

As components are combined, the constraints can be propagated - creating new constraints that detect when the higher level components violate the lower level constraints or any of their own constraints. Once the constraints propagate to the top level, they can be used as a measure of the system's correctness against the specifications.

It is not necessary that the constraints be determined by static analysis or that mutation testing is used to test the constraints. In some cases, static analysis may be more effective at determining a constraint violation (for example, order of evaluation errors), or the programmer may have to assert constraints for more complex adaptability errors (for example, protocol dependencies) via a PRAGMA or by some other means. In
any case, this approach is promising because it limits the amount of testing needed for the reusable component to those areas where a known potential for a reuse error exists.

This approach is an improvement over the strictly static approach in that it deals with the two-point error issue by recording the potential errors at module implementation time and because it detects actual reuse errors at component integration time—without completely re-analyzing the components. It is also an improvement over mutation analysis because the execution overhead is reduced. A component would be mutation analyzed for standard errors but the problems with inter-procedural errors (reuse errors) is resolved by limiting the scope of the testing to those areas where known potential for errors exists. Constraint propagation helps not only in guiding a hierarchical testing method, but can also be used for automatic generation of test data that exercises the constrained paths.

CLASSIFICATION OF PORTABILITY ERRORS

Porting software between machines is a type of software reuse that is well-understood and has been much described (Wallis, 1982; Nilssen and Wallis, 1984; Barnes 1984). Less well understood are methods of preventing these dependencies and of correcting them when they are detected. Presented in this section are several ways in which portability issues can be characterized to help software developers prevent or detect them.

Portability problems vary depending on the application domain or development method in use. When a multitasking application is developed, timing and synchronization considerations become an important concern. Numerical applications may require that special attention be given to the precision and correctness of the environment’s arithmetic functions. Issues characterized in this manner can be used to program defensively, to detect occurrence, or to document what cannot be avoided.

Another way to look at portability issues is the manner in which they
can be detected. Issues associated with a particular statement or a particular program fragment can be detected with static methods. The presence of an UNCHECKED CONVERSION PRAGMA can be detected in such a manner and indicates a nonportable statement. A function that modifies a global variable has a potential dependency on the form of the expression used to call it. However, this situation can be detected statically. Other issues require execution of the program to detect the problems caused by an active process interacting with and changing the state of its environment.

A third way to look at portability issues is based on the Ada constructs used. The language designer or someone learning a language studies it feature by feature. Manuals or development tools point out undefined or implementation-dependent features. The Ada Language Reference Manual (Department of Defense, 1980) tells which properties are undefined and includes an appendix describing an implementation's machine dependencies. The remainder of this section looks at portability problems from these three viewpoints.

Types of Application

This categorization is based on the assumption that it is better to avoid constructing non-portable code than to correct or remove a non-portable construct after it has been written. The person who best knows a piece of software, and thus who can best prevent or document non-portable sections of code, is the one writing the software. As software is developed, decisions are made that affect its portability. A decision may introduce new concerns, or alleviate others. For example, a programmer decides some numerical computations are necessary for an application. Now he must worry about such things as word size, how the machine handles overflows, and the accuracy with which the machine and runtime system implements arithmetic operations. If he then decides that the data will never exceed some small value, his decision eliminates concerns about word size and overflow.

Some portability issues are a concern no matter what the type of application. For example, the way a compiler compiles the source, the optimizations used, the algorithms chosen, all affect the space the object code occupies and its execution time. A program that executes on one machine may not be able to fit on another machine that has the same amount of space, when compiled with a different compiler. Any program that inadvertently accesses a variable before it has been assigned may become dependent on the memory initialization policy. Most issues, however, are related to the application and method of development. This characterization indicates when a programmer developing a piece of software must worry about a particular portability issue.

Numerical. When an application is primarily concerned with numerical computations, issues such as convergence and error propagation arise. For example, when numbers become larger than the machine can represent, some machines signal overflow while others continue silently with an incorrect result. Other concerns include whether the format of numbers is one's complement or two's complement, whether floating point numbers that cannot be represented are rounded or truncated, what is the precision available for floating point numbers, and how bytes are aligned (Spafford, 1988).

Multitasking. A programmer coordinating the execution of several Ada processes has a difficult task in writing portable code. Two compilers may produce differing object programs for the same source, even if they reside on the same hardware and operating system. The timing of these object programs will differ, and so coordinated processes may not interact in the same manner when compiled differently. Unsynchronized use of shared
variable is a portability issue when a programmer develops coordinated processes. Other problems arise when an implementation permits the use of machine priorities, as priority levels may differ between machines.

**Dynamic memory allocation.** Some applications make heavy use of dynamic memory allocation. Structures that grow and shrink dynamically can outgrow the space available to a program. Machines differ in the amount of space they allow for dynamic allocation. The method of allocating pointers also differs between machines. A program that depends on a particular allocation strategy must indicate this, as its portability will be affected (Spafford, 1988).

**Systems programming.** A programmer may decide to specify the way an object of a particular type is represented, the maximum space to be used to store the object or its storage location. This may be done to tailor the code for a particular machine and obviously will not work on other machines. However, it may be an attempt to parameterize the code so that it will behave properly regardless of the underlying machine. For example, if a construct may be represented differently on other machines, it may be represented in terms of machine-specific symbolic constants. Even though this can make software more portable, another compiler may not be able to represent the construct as specified, thus making the software nonportable. Unchecked conversions also depend on a machine's representation and may not be portable.

**Detection**

A characterization based on prevention can help in the development of portable software. However, it doesn't say anything about detecting potentially non-portable sections of existing code. This subsection discusses how the various detection mechanisms apply to different types of portability problems.

**Static analysis.** Certain features of a language are potentially non-portable and can be detected simply by searching for them in the code. These include specifications for the representation of complex types, unchecked conversions, unchecked deallocation, machine code insertions, and use of machine constants. The presence of any of these doesn't necessarily mean that a portability problem exists, just that the potential for a problem exists. The potential for other portability issues, such as a dependency on a particular order of evaluation or on a particular memory initialization policy, can also be detected through static analysis.

**Simulation.** Portability issues that stem from a dependence on the way the environment and the executing program interact cannot be detected by statically examining the code. These portability issues depend on the way the environment does such things as allocating and laying out storage, passing parameters, and elaborating library units. The synchronization of multiple processes is highly dependent on the manner in which the processes interact with the environment. The interaction must be observed to detect such dependencies. Problems associated with these issues can be detected through executing the software on simulations of different environments.

**Mutation analysis.** Other portability issues lead to problems that can be detected through manipulation of input. Still others vary from machine to machine with a small number of options, such as whether the machine raises an exception on overflow or continues silently with an incorrect value. These issues include whether a machine rounds or truncates values that have no representations and how it handles operations on singular values (e.g., negative zero on a one's complement machine). Mutation analysis detection mechanisms that can simulate specific environments can
Constraint analysis. Despite a developer's best attempts at making a piece of software portable, a program may still have minimal requirements that an environment must fulfill for the program to execute properly. For example, numerical software may require at least a certain precision for floating point calculations. A program may need an unusual amount of memory available for dynamic allocation. Constraint analysis can be used to develop a set of constraints that must be satisfied by an environment to execute the program.

Language

A third way to characterize portability issues is based on the language used. Portability issues are categorized by the language constructs in which they arise. The portability of a piece of software can then be described in terms of the constructs used and the problems associated with those specific constructs. Naturally, a programmer who is trying to develop portable software can utilize such a characterization when deciding what constructs to use.

Some of the books written about software portability talk about portability in terms of the various language elements. In particular, Nilssen and Wallis (1984) describes portability issues for Ada by pointing out potential problems in each section of the Ada Language Reference Manual (Department of Defense, 1980). The manual itself mentions when the implementation of a feature or an algorithm is undefined and therefore varies between different machines.

Summary

The language-based and the application-based characterizations can be used to alert developers of portability problems inherent in the decisions they make about the use of particular language constructs and in the type of application being built. Developers can be guided in documenting issues that have not been resolved so that nonportable sections of code can be easily located and modified when necessary. The characterization based on detection can guide measurements of software portability. It also defines the limitations of the detection mechanisms.

CLASSIFICATION OF ADAPTABILITY ERRORS

Software adaptation is the process of moving a reusable software component from one application environment to another. Adaptability errors are anomalies that cause a reusable component to fail to behave correctly when used in an application environment for which it was not designed.

The software adaptation process involves the selection and retrieval of components from a library of software parts. Such a process presumably would be based on the Ada compilation unit structure and would utilize packages and generics to group components and allow for their customization. Furthermore, some sort of database would be available for selecting an appropriate component based on its specification. It is up to the application developer and this system to determine the suitability of a reusable component for a given application. To realize a cost savings in reusing a component, it must be possible to make such a determination from a component's specification because the cost of understanding source code is high.

Ada goes beyond most other languages in its separation of
specification from implementation. Moreover, it provides encapsulation mechanisms that encourage the production of modules with strictly limited interactions with their invoking environments. In an ideal world, the Ada specification mechanism would provide a complete description of all dependencies between a module and its environment. In such a world, all reuse errors would be detectable as violations of the specification, and traditional V&V mechanisms could be used to detect them.

There are several areas in which Ada's specification mechanisms are insufficient to describe important interactions between a reusable component and its application environment. In the first area, responsibility for providing certain auxiliary capabilities, such as concurrency control, is difficult to describe in an Ada module specification. In the second area, the language gives too much freedom to the user, and abuses can lead to subtle reuse errors, such as when two arguments to a procedure are aliases for the same variable in the calling environment. The next two subsections describe these two areas.

Architectural Concerns

When a software system is designed, there are typically areas of concern that go beyond the strictly functional specification of the module. For example, if dynamic memory allocation is being used in a software system, whose responsibility is it to manage/free allocated objects? It may be the local responsibility of the module that provides the objects, or it may be the responsibility of a general-purpose memory management component. In either case, it is important that the responsibility be explicitly specified.

This section gives examples of four types of architectural decisions regarding the responsibility for managing certain capabilities. Grady Booch (1986) presents these issues in the following classification. E. V. Berard (GRACE, 1987) has presented a similar list of general issues that involves greater detail and issues that relate to the time and space performance of a component. Both Booch and Berard sell collections of components organized around these decisions.

Concurrency. Components are designated as one of sequential, guarded, concurrent, or multiple. Each of these categories describe a different level of client application concurrency control. Sequential allows only one thread of control; guarded requires the client to enforce mutual exclusion; concurrent provides mutual exclusion; and multiple allows multiple simultaneous readers.

Space Utilization. Components are rated as either bounded or unbounded, which describe an object's space allocation scheme. Unbounded objects can grow dynamically as long as memory is available while bounded objects have some programmer-imposed maximum size.

Space Reclamation. Components are classified as unmanaged, managed, or controlled, each of which describes a different level of garbage collection control. Unmanaged components rely on the machine environment to do garbage collection; managed components perform their own garbage collection; and controlled components perform garbage collection even for multiple client tasks.

Iterator Availability. Components are rated as one of iterator or noniterator, each of which indicates the availability, or lack of availability, of an iterator. An iterator is an operation associated with an abstract data type that yields the elements of an object of that type, one at a time upon request. Noniterator components have no iterator, and
iterator components do have an iterator. Iterators are generally used with
looping constructs to perform an action on each element of a data structure
without exposing the implementation of that structure.

According to Booch, reusable components should be classified along
each of these dimensions, and this classification should be made part of
the component specification. Thus it is an adaptability error for an
application to use a component, assuming one classification, when in fact
the component does not meet that requirement.

For example, assume there is a component that implements a priority
queue that is to be used by an application. This component may have been
constructed with the assumption that only a single program or Ada task will
be executing the code at any given time, but the application may require
the component to allow multiple threads of control and manages the
concurrency with an appropriate set of controls. In such a situation, it
is highly likely that an adaptability error will occur. Here the
application environment places a demand on the component that is unrelated
to the functional properties of the component. How a component is designed
with respect to these issues must become part of all reusable component
specifications in order to deal with this type of adaptability error. A
V&V method for reusable components should include methods for verifying
that a component meets this part of its specification. Some reasonable
classification similar to Booch's makes an appropriate addition to standard
functional specifications.

Implementation Concerns

The issues presented by Booch and Berard are fairly general in nature
and involve features that often must be designed into a component. Dealing
with these problems is a matter of recording this classification in the
component specification and considering this information when the component
is selected. Even though a component may be properly characterized along
each of these dimensions, there may be the potential for improper use of
the component that will lead to an adaptability error. Taking the example
above, the priority queue component may be properly specified as allowing
multiple threads of execution but may require the application environment
to use a provided set of routines in a protocol that assures mutual
exclusion, such as a graph-based locking protocol. If the application
environment fails to use the protocol properly, the component may fail.

This type of error is very similar to those mentioned above except
that it is much more component-specific and represents an error in the
application environment, as opposed to an error or omission in the
component's specification. Besides being more local, these problems relate
more closely to the program's implementation than to its design. Moreover,
they reflect instances where the semantics of Ada offer the programmer
subtle opportunities to get into trouble. For example, aliasing problems
arise because it is possible to give two or more names to the same element
of a program's state.

These errors can also be taken as a measure of a component's
robustness, its ability to deal with improper or unexpected input. For
example, a functional formal parameter to a generic unit must have its type
as well as the types of its arguments specified. However, other important
properties, such as whether it is commutative are not explicitly mentioned.
The following is a list of some of the areas where this kind of
adaptability error can be found:

Order of evaluation errors. A component may export an object and a
function that modifies the object. If the application environment uses the
object and the function in the same expression, on opposite sides of the same assignment, in argument expressions for the same procedure call, in two index expressions for the same ARRAY object, in components of the same aggregate, within the same range expression, or as guards in the same SELECT statement, then the potential exists for a dependency on the order of evaluation.

**Aliasing errors.** A component may export an object and a procedure that takes an IN OUT or ACCESS argument of the same type. If the application environment calls the procedure with the object as the argument, then the potential exists for the object to be referred to via two name within the component. Similarly, the problem may arise if the component exports a procedure that takes two IN OUT or ACCESS arguments of the same type, and the application environment calls the procedure with the same object for both arguments.

**Domain errors.** A subprogram's behavior may only be defined if an argument or object's value is within some range or bears some relationship (>, <, =, etc.) to some other argument or object's value, and the application environment invokes the subprogram when the condition is not met. In many cases Ada provides mechanisms such as subranges for dealing with this class of problem. There will always be instances, however, where a component requires more of its parameters than the language permits it to specify. The case of the formal generic functional parameter described above is an example.

**Protocol errors.** The specification of an Ada component places requirements on the parameters of the component when it is used. The specifications, however, are restricted to describing a single invocation or access to the component. If the component retains some internal state between, then there may be some limitations on the legal sequences by which the component may be invoked. For example, a stack component should never have more POP calls than PUSH calls. Although these situations can be checked for explicitly at run time, there is no static specification mechanism for dealing with them in Ada. If the component relies on the application environment to maintain such constraints, then the potential for a reuse error exists when the component is moved to a new application environment. These problems are called protocol errors. Another example occurs if a component exports two subprograms and the number of times one is called may not exceed the number of times the other is called by more than N times (as would be the case in a bounded implementation of a queue). There are numerous examples of protocol errors such as the use of semaphores and initialization of sub-systems. Any component that has some notion of state may be susceptible to protocol errors.

The common thread among these categories of implementation concerns is that the component in question is designed in such a way as to allow a client application to misuse the component. There exists the potential for an error. The realization of the error exists somewhere across the application interface in the application environment itself, otherwise the error is either a portability error (as in the case of aliasing, order of evaluation, etc.) or a traditional error (as in the case of a domain error). For this reason, traditional V&V techniques are of little use, since there is not enough context to determine the existence of an error, regardless of the computational power available. A possible solution is to utilize constraint-based analysis to develop usage constraints that can be stored with a component's specification and utilized by an integration testing tool to test a client application for compliance with the constraints. Such a system could use a variety of techniques (e.g., static analysis, mutation testing) to both determine the constraints and test for their compliance. Complex constraints, such as usage protocols might be
specified by the programmer as assertions included in the specification.

ISSUES

Objective

Traditionally, the goal of V&V is to reach a point in the software development lifecycle where it can be stated that a component is free from errors. In practice, the best that can be achieved is to determine that a component is free from a restricted class of errors. This often involves assuming the component, after design and coding, is relatively close to being correct, as discussed in DeMillo (1978).

In a similar fashion, it would be desirable to determine that a component is free from reuse errors. Due to the wide range of portability and adaptability reuse errors, however, this goal must be limited to a restricted class of reuse errors. The objective then is to develop a notion of adequacy for a restricted class of reuse errors that aids developers in the V&V process in developing and utilizing reusable components effectively. Such a notion should help to limit the combinatoric explosion involved in testing a system built from reusable components.

Adequacy and Reuse Errors

In attempting to develop a testing method to determine the correctness of a component, it is desirable to have some sort of adequacy criterion. An adequacy criterion for a class of errors is a condition such that if a test data set satisfies the criterion, then the test data set is capable of detecting all of the class of errors that might exist in a component (Weyuker, 1983). In essence, an adequacy criterion is a mechanism for determining when component testing can be stopped. An example of an adequacy criterion is "the test data will cause each and every statement in the component to be executed at least once." There is, however, no guarantee that a test data set that satisfies an adequacy criterion will be effective in detecting other classes of errors (Weyuker, 1988). The well-known statement coverage criterion mentioned above is deficient at discovering many detectable traditional errors (Walsh, 1985) but may be used in combination with more powerful criteria such as those developed through mutation analysis (DeMillo, et al., 1988) to produce effective adequacy criteria.

In dealing with reuse errors, the first problem is to understand what it means to be free of reuse errors. Previous sections of this paper have discussed what reuse errors are, but it is not clear that it is even possible to develop components that do not have reuse errors. For example, almost any component has at least a few arithmetic operations. Since arithmetic can be highly machine dependent, it may be impossible to write a completely machine-independent component. Similarly, it can be difficult to develop programs without introducing adaptability errors such as aliasing errors.

Therefore, the goal should be to develop adequacy criteria such that if a test data set satisfies the criteria for a component, then when the test data set is executed against the component in some environment, the test data set will detect any errors that exist in the new environment, rather than criteria that simply determine the existence of reuse errors in the isolated component. In contrast to traditional adequacy, that was relative to a set of errors, reuse adequacy is relative to both a set of errors and a target environment. This is achieved by describing the test
Figure 4. A typical system with reusable component B.

data set in terms of the environment, and should be reflected in the component specification.

The bulk of this research has been concentrated on understanding the class of reuse errors. Reuse errors are more complicated than traditional errors because software reuse implicitly involves a wide variation in environments. Portability issues are complicated because the semantics of even the most basic machine operations such as arithmetic are not uniformly defined. Adaptability is similarly complicated by the total freedom an application environment has in using a component or supplying dependent modules to the component that may or may not perform as the component expects it to.

Conclusions

Figure 4 depicts a typical software reuse situation, where B is a reusable component, A is that part of the environment that depends upon (or calls) B, and C is that part of the environment that B depends upon (or calls). As an example, A could be a database application; B, an Ada generic implementation of a binary search tree (BST); and C, the Ada runtime system, operating system, and hardware, and the function '>' that B uses to create a total order of the elements in its data structure. This function is supplied as a functional argument in the instantiation of B.

B has two interfaces to its environment: its interface to the system, the machine interface, and its interface to the application, the application interface. These two interfaces can be separated by redrawing the diagram as in figure 5, with the A to B interface representing adaptability and the B to C interface representing portability. Figure 5 notes that a portion of A, the application environment, should really be positioned as depended on by B (as the function '>' would be in the example), and a portion of C, the machine environment, should really be positioned as dependent on B (as would be the case if B was responsible for management of dynamic memory). These additional dependencies are indicated by the use of braces.

Portability Issues. Consider the effects of portability errors in testing B. Because B calls C and reacts according to the results of C, B's correctness depends on the semantics of C. In particular, all of the portability errors discussed above arise when B assumes the semantics of C
Figure 5. A typical system showing adaptability and portability.

to be different than they are. The obvious solution to this problem would be to design B such that it does not depend on C at all, but this is entirely infeasible, as even the most fundamental machine characteristics, such as arithmetic, vary considerably from machine to machine. As new machines are designed, the possibility for new portability problems are compounded.

The dilemma can be stated as follows. Traditional black box testing measures correctness by looking at the behavior of a component, such as B above, and everything that it relies upon. But these dependencies are beyond the control of the designer of the reusable component. Consider the example of the generic Binary Search Tree: if B (the generic) is provided a function '>', that implements a less-than function (instead of a greater-than function expected), then the behavior of B may not match its specification. Nevertheless, there is nothing wrong with B; B's correctness depends on the validity of its functional argument.

The problem with portability errors is that components are specified in absolute terms, regardless of the machine environment. In order to deal with portability issues in correctness, specifications must be at least partially dependent of the machine environment. The dependent part of the specification can be parameterized: the behavior of component B expressed relative to part of the machine environment C. With a parameterized specification, it becomes possible to create a well-defined virtual machine so that all specifications and subsequent testing can be performed in terms of that machine. A testing environment then would provide the capability to test a component's dependence on this environment at the places defined by the portability error characterization.

Finally, it is worth noting that Ada, does not provide adequate support for specifying components in this fashion. Support is provided for coding Ada parameterized components via machine attributes such as Machine_EMax and Machine_Overflows, Storage_Size and subrange attributes such as First and Last. Ada specifications, however, do not constrain the implementation of components to use these parameters beyond the use of
Adaptability Issues. Now consider the effects of adaptability errors in testing the system A B C with an adequate set of test data, T. Weyuker (1988) states in her anti-decomposition property of adequacy criteria that if "there exists a program P and a component Q such that T is adequate for P. T' is the set of vectors of values that variables can assume on entrance to Q for some t in T and T' is not adequate for Q." This means even if A B is adequately tested, B has not necessarily been adequately tested. Weyuker also states in her anti-composition property that "there exists programs P and Q, and test set T such that T is adequate for P, and the set of vectors of values that variables can assume on entrance to Q for inputs in T is adequate for Q, but T is not adequate for P;Q." This means isolated unit testing of B is insufficient to detect errors that arise from the combination of A and B - which are exactly the adaptability errors described above.

Therefore, reusable components cannot be adequately tested for reuse errors within some existing application environment, nor can they be tested without any application environment at all. Instead, some sort of virtual application environment must exist to test the component that can be systematically altered to effect the simulated adaptation of the component, thereby exercising the potential reuse errors of the component. This result bears a comfortable resemblance to the well-defined virtual machine for testing portability issues, but may not seem as obvious.

Testing under such a virtual environment amounts to the development of explicit constraints on the use of the component B, defining those application implementations that would result in the incorrect use of B. These constraints are then used as a kind of negative specification against any target application environment A, thereby limiting the testing of A to exactly those areas that are known to create problems when A is used with B.

Further Work

The research conducted to date has been concerned with understanding the nature of the problems involved in V&V for reusable Ada components. Specifically, it has concentrated on looking at the principle reuse concerns, adaptability and portability, how they are manifested in the Ada language, and at developing an understanding of adequacy for reuse errors.

Two areas of further work look particularly promising for making the results of this investigation useful in a practical sense. The first is the development of tools. A prototype analyzer for Ada programs has been constructed as a means to detect certain reuse problems. This tool should be extended via simple source code analysis techniques to detect a wider class of problems. Furthermore, this tool can become the basis for a V&V extension to an Ada development environment for determining adaptability constraints. These constraints would be stored in the Ada component library, and utilized for generating specific tests when the component is used in an application (figures 2 and 3).

The second area concerns the relation between white box and black box testing. Problems with adapting a component to a new environment involve the functional (black box) behavior of the component. Most of our tools, (Ada compiler, Mothra, the prototype Ada analyzer) use white box techniques. We would like to use the black box specification of the component's behavior to drive a white box analysis. Specifically, we would like to improve the detection of reuse errors and facilitate the generation
of test data by making direct use of the program's functional specifications.

BIBLIOGRAPHY


