The Testability of Ada Programs

David Auty, SofTech, Inc.
Norman Cohen, SofTech, Inc.

Software development for NASA's space station poses a significant challenge; considered the most difficult challenge by some. The difficulty is the magnitude and complexity of the required software. With the requirements for remote control and communications, software will lie at the heart of many essential and complex systems within the station. The combined requirements for highly-reliable systems exceed any software development effort yet attempted.

NASA's previous experience with software development centers on the assembly code and the code in the high-level language HAL/S, developed for the space shuttle. Within the development of that software there was heavy reliance on careful testing and thorough multi-level checkout. Within the HAL/S development environment, the checkout procedures could depend on the stable characteristics of and limitations on program behavior inherent in the language. This paper addresses the concerns raised by consideration of the requirements for testing and checkout procedures for the space station software. In particular it addresses the use of Ada in the development of widely distributed yet closely coordinated processing.

This analysis is done in two contexts. First, an evaluation of the language is presented, discussing how the rules and features of the Ada language effect the testability of software written in it. Second, some general techniques in software development which can augment testing in the development of reliable software and some specific recommendations for tools and appropriate compilation are presented.

This paper is a summary of a full report prepared at the conclusion of an extended study effort on this topic. It therefore does not go into detail in elaborating each point of interest. An attempt has been made to cover the breadth of the report and present its key findings.

Evaluation of Ada

We begin by discussing how a programming language can be evaluated for testability. For our purposes, testability is the ability to determine, by test execution of software, whether the software will function correctly in operational use. Testability measures the extent to which it is possible to construct tests such that the behavior of the software on those tests reflects

the behavior of system when deployed. Among the issues related to testability are the ease of generating comprehensive test cases, the predictability of resource utilization under all circumstances and the deterministic repeatability of processing sequences.

This definition applies principally to the developed program, but it can be extended to apply to the language used to express that program. A programming language supports testability to the extent that it facilitates the writing of testable software. We have identified the following attributes of a programming language which facilitate testability:

- support for modular decomposition (i.e., supporting the testing of units independently of their use in the system),
- existence of interface specifications constructs which are clear and comprehensive
- complete type and program unit specifications allowing comprehensive consistency checking during program compilation,
- well-defined run-time error handling,
- predictable resource allocation and utilization,
- support for the writing of test drivers and hardware stimuli simulation, and
- support for the creation of high-level abstractions.

With these evaluation criteria, we considered the following aspects of the Ada language:

- Data Types and Subtypes,
- Separate Compilation and Packages,
- Subprogram Definition,
- Generic Units,
- Exceptions,
- Concurrent Processing and
- Storage Management.

Each aspect was considered from the viewpoints of conformance with evaluation criteria, risks to testability and recommendations for reducing those risks.

Fig. 1 shows an evaluation criteria versus features matrix showing the extent of support of the Ada language for testability. The matrix shows where aspects of the language support the evaluation criteria, independent of the possible risks within the same feature area. In general, the strong typing rules of the language and the concept of separate specification and program unit bodies provide excellent support for testability.

	Data Types and Subtypes	Separate Compilation and Packages	Subprograms	Generic Units	Exceptions	Concurrent Processing	Storage Management
Modular Decomposition	●	●	●	●		●	
Clear & Comprehensive interface specifications	●	●	●	◐	◐	●	●
Compile time consistency checking	●	●	●	●			
Well-defined run-time error handling	●				●	●	●
Predictable resource use and allocation	●	●	●			●	●
Support for test and test driver programs		●	●	●	●	●	
Support for creation of high level abstractions	●	●	●	●	●	●	●

Fig. 1, Evaluation Criteria vs. Feature Matrix

Two areas of particular interest are represented as only half-filled circles in the evaluation matrix. These represent qualified support for the evaluation criteria. In the case of exception management, the rules for the raising of exceptions, including user specified raise statements, and for exception propagation, allow for a very concise treatment of exception processing. Thus, when properly documented, exception processing as defined in the language is an important part of a module's interface, supporting the requirement for clear and comprehensive interface specifications. Because it is dependent on optionally included comments, however, this can be considered only qualified support for the evaluation criteria.

The second half-filled circle is under generic units. This is a similar situation as for exceptions. The rules for formal generic parameter specification and for generic instantiations allow for a clear and concise specification of the units interface. However, as will be discussed under risks, there are secondary aspects of actual parameters (which we term second order properties) which are not documented, such as functional requirements on actual procedure parameters. Because these secondary aspects can be critical, yet possibly undocumented, support in this area is also qualified.

Testability Risks

In the evaluation of the Ada language features, several risks to testability as well as the above benefits were identified. These risks fall into two broad categories of inefficiency and hidden interfaces, plus one additional concern without such convenient categorization.

The concern over efficiency is based on a simple assumption that features which fail to provide adequate efficiency will not be used in many applications. The resulting program which may be more or less convoluted in its avoidance of this feature will certainly not have benefited in its testability. Although processing capabilities and memory sizes are increasing dramatically, the requirements to surpass the increased capabilities are already being considered. Concerns over efficiency in Ada fall into three areas:

- excessively expensive run-time checks,
- inappropriate or undirected instantiation of generic units, and
- excessively expensive tasking architecture.

These can be collected under the general concern of inefficiency in support of high-level abstractions.

The second broad concern is that of hidden interfaces. Despite the strong support in the language for detailing important interface information, several possibilities for hidden interfaces exist. Hidden interfaces exist wherever interactions or dependencies exist which are not part of the specification or declarations of the unit. These can be classified as being due to:

- global variables (side effects of procedure and function calls, contention over access between separate tasks),
- the raising and propagation of exceptions,
- dynamic storage utilization,
- dynamically determined timing behavior, and
- second order properties (e.g. functional requirements on actual procedure parameters) for generic instantiations.

An example of second order properties would be the case of a generic sorting procedure. A typical implementation will have the type of the objects as a generic parameter, requiring a second parameter to be a function which can compare values of that type and return a boolean value on the basis of the condition "less than". The second order property of the actual function used during instantiation is that it must return a proper ordering of all values of the type. In fact, it is conceivable that the sorting routine may never reach

an exit point if the function does not have this property. Yet this property is not required in any way by the language during instantiation.

One last risk for testability is the general non-determinism of tasking interactions. While not so much a fault of the language, as asynchronous concurrent processing is inherently non-deterministic, the presence of tasking in an Ada program can complicate the testing of that program.

Recommendations to Reduce Risk

In response to the identification of these risks, several recommendations for reducing the risk were made. These fall under the general headings of:

- requirements for appropriate development practices and training,
- requirements for appropriate tools, and
- requirements for appropriate compilation.

The principle behind the requirements for appropriate development practices and tools is based on the recognition that their use can help assure reliable software where testing is difficult. Testing practices can be augmented by the use during development of proof techniques, static program analysis and runtime monitoring. Throughout the development process, verification techniques can be used to insure principles identified and verified early in the development are held true through implementation.

For appropriate programming guidelines and training, the following suggestions were made:

- For numeric processing, training should include a discussion of digital computation algorithms and their interaction with underlying numeric precision in determining the accuracy of the computed value. This is necessary to put the rules for numeric precision of the language in proper context.

- Programming Guidelines should be established for:
  - the judicious use of suppress and inline pragmas to provide efficiency as necessary,
  - the avoidance of global variables and hidden side effects,
  - the hiding of persistent variables in package bodies (and therefore private to the package), and
  - the use of out parameters from procedures over unconstrained composite results from functions (allowing better storage utilization).

- Training should emphasize:
  - concurrent programming concepts and practices
  - the concept and significance of second order properties of generic parameters

- Standards (with enforcement) should be established for:
  - the documentation and use of exceptions
  - storage utilization practices

A more reliable approach to improving testability is through the use of appropriate tools to aid in the development process. The following are some tools to specifically address the risks for testability identified:

- Proof systems for verifying 2nd order assertions in generic instantiations and assertions about task interactions, task state systems and other program properties.
- Runtime monitors for deadlock and other deadness errors, storage utilization parameters, and other runtime properties.
- Static program analysis for tasking interactions, storage utilization and other program properties including adherence to the programming guidelines listed above.
- Expert system support such as a "real-time assistant" for cyclic-based system generation.

Having identified program efficiency as a risk to testability, in that good features of the language will not be used if they are not sufficiently efficient, several suggestions for appropriate compilation should be considered. In general, a highly optimizing compiler, with efficient, deterministic runtime support is a necessary goal. Particular attention should be given to the following features:

- optimization of subtype range constraint checking,
- reduction of uncertainty in the raising of predefined exceptions,
- space efficient compilation with pragmas and representation clauses for user control of storage utilization,
- optimization of tasking interactions with special support for tasking paradigms through pragmas or pattern recognition, and
- efficient size and speed of generic instantiations with pragmas for user specification of instantiation criteria.

Summary

In summary, it was found that the language offered the potential to greatly improve the testability of software, provided that certain guidelines were followed. The language introduces features to deal with higher level abstractions and the complexities of concurrent processing and dynamic storage utilization. These features are considered necessary to deal with the complexities of the space station software requirements, but can decrease the testability of that software. These risks to testability can be dealt with through a combination of appropriate development practices and training, appropriate tool support and appropriate compilation.