N89-16332

CERTIFICATION OF ADA PARTS FOR REUSE

Gregory A. Hansen*,
General Dynamics, Data Systems Division, San Diego, CA

S.D. Spaulding, General Dynamics, Data Systems Division, San Diego, CA

Glenn Edgar, General Dynamics, Data Systems Division, San Diego, CA

* Currently employed by the Software Engineering Institute,
Carnegie-Mellon University, Pittsburgh, PA

Introduction

One of the claims made by proponents of Ada is that Ada software is highly reusable. The fact that specifications are compiled and accessible would make reusability seem easily achievable. However, specifications give only a limited amount of information about a package; moreover, a specification cannot help determine whether a package "worked", or how well it worked.

This problem has led to the concept of "certifying" Ada parts for reuse; that is, determining the worthiness of a part as a reusable component. This paper addresses issues that are critical to reuse: the characterization of part performance, design for reuse, and correct utilization of parts. The paper will then address current areas of study beneficial in the development of a certification process.

I. Characterization of Part Performance

Ada has two features which support reusability: specification compilation and the ability to declare instantiations of generic units. Ada specifications allow a part's interfaces to be defined, but are not sufficient for defining reuse. It is important to have information about a part's performance in a reuse operation to determine computational requirements, accuracy of calculations, etc.

Performance is not something that is easily quantified; however, attributes associated with performance are definable. A part's performance is definable by its behavior, or intended function, and the computational resources it extracts from the system when executing. These attributes are directly related, and not independent. They must be considered in the scope of both normal processing and exception handling.

The explicit separation of exception handling and normal processing is essential for modularization. Without exceptions, nested flags are required for error recovery management. This leads to the intermingling of error (i.e. exceptions) and normal processing, which leads to unmanageable code. Ada provides for the separation of exception handling and normal processing, and this separation is mandatory for parts reuse.

Exception handling consists of three steps: exception detection, correction, and recovery. These steps should be handled at different places in a software system. The exceptions that are raised, and the method of handling those exceptions, are not contained in a package specification. This information is essential to the part certification process.

How exceptions are handled determines the behavior of an Ada part and affects the performance of that part. The detection, correction, and recovery philosophy of a system has direct bearing on the computational requirements of that system, as does frequency of exceptions. Subjects such as recovery vs. restart and process synchronization must be addressed. Exception handling standardization could be an important factor in the certification process.

GENERICS.

Generic programming seems to provide a logical approach to the certification of reusable software. However, some obstacles to reusability, such as exception handling, still exist. Specifications for generics give no more information concerning the behavior or performance of the corresponding body than specifications of packages, subprograms, or tasks. Since Ada does not allow exceptions to be passed as parameters for the instantiation of generic parts, the use of a generic as a reusable part is somewhat constrained. The usual exception declaration interface between a system and a generic package is the package specification. Thus the identity and meaning of the exception is determined by the generic package, not the host program. This constitutes a reversal of accepted top down design techniques.

Another way of interfacing exceptions and reusable generic parts is to have both the system and the generic unit depend on a specifaction package of exception declarations. This technique would be consistent with top down methodologies, but would require a high degree of cooperation between system implementers and the designers of reusable parts.

A third technique of exception interfacing involves the implementation of subprograms which raise exceptions. The subprograms would be elaborated in the system's declarative environment and passed as actual parameters to generic instantiations. This technique solves the problem, but at the cost of efficiency, elegance, and design clarity.

CONSIDERATION OF TIMING

Another factor affecting certification of reusable parts is the hard timing requirements of a part. This information is not extractable from a package specification, and varies from system to system. In early computer architectures, timing was a fairly easily calculated quantity. However, multitasking software systems and new architectures which use cache, floating point accelerators, and other features, have direct influence on timing. In fact, non-package constraints such as context switch times have become as important as package timing itself.

Since parts can be viewed as tree structures with many branches; where exception handling and timing must be considered, the characterization of a part's performance and its certification are indeed very complex.

II. Constraints in Design for Parts Reuse

The design of software parts must be done in a context independent manner; that is, no assumptions should be made about input conditions. All possible error conditions should be anticipated and treated as exceptions. The exception handling implementation must be explicitly documented.

The Ada compiler run time default error checking features should not be used, except as a redundant check. If run time error checking is turned off for speed reasons, then flaws potentially exist in the system. Therefore, error conditions must be handled by the package. This philosophy, unfortunately, can lead to speed impacts within the system.

If there are time constraints placed upon a part, then a "cost" analysis must be performed on that part prior to its implementation, and the results of that analysis must be captured for later use. A hierarchical funtional decomposition methodology, such as data flow or Petri nets, can be used in the analysis process. As will be discussed later, expert system technology can be applied to the performance of the "cost" analysis.

It should also be mentioned that there exists a potentially large number of specific coding and design practices that can adversely impact reusability at both the system and part level. To fully identify these practices and address their relative impact will take time and experience, and such a discussion is beyond the scope of this paper.

III. Certification and Reuse

Ideally, a certified "part" should be a reusable part. However, it is probable that parts that are considered to be 100% certified are going to be small segments of code with limited application. The process of certifying large segments of code is extremely complex.

This paper has made several points concerning the reuse of Ada parts:

- o Ada specification packages are insufficient for determining reuse
- o Behavior and performance of a part must be explicitly defined and extractable
- o Exception handling is an important factor in both behavior and performance
- o Generics offer a logical approach to certification of reusable parts but have certain constraints
- o Hard timing requirements must be stated, and are subject to variations created by hardware and software environments
- o Run time implementations must be considered as influencing a part's behavior

Artificial Intelligence can provide some technology to reduce the complexity of analysis for reuse. In particular, expert system technology and object-oriented design can be applied to the problem. Object-oriented design is a term used to define a methodolgy of software development in which data items in a software system are defined in terms of their attributes, as well as in terms of their relationship to other data items in the system. Object-orientation has led to the concept of "frames", which are used extensively in expert systems for knowledge representation. If software parts are thought of as objects, a frame-based system can be built which contains declarative and procedural information about parts.

The knowledge contained in such a frame would be symbolically stated, using a formal grammar. The grammar of the frame will state the function of the part, such as number and types of exceptions, real-time requirements, accuracy, etc. If a hierarchical representation is used to describe the system, attributes of parts can be "inherited" from other parts at a higher level in the hierarchy. An expert system can then be built to compare requirements to information about parts, yielding a probabalistic measure of applicability of a part to a problem. The more information available about a part, the better a measure of applicability can be determined.

Another technology that can be applied to reusability is that of Archetyping. (1) Archetype comes from the latin archetypum for "first molded as a pattern; exemplary". In this case, software specialists capture software soon after it has been tested and delivered, work with users of the software system, and sketch out future requirements for systems of this type . Thus, a team of software and domain experts develops a pattern from which future systems can be generated. The result of an archetyped software system is a template that requires a tool to "fill in the blanks" to customize the software for an application. One such tool is the DARTS technology, offered by General Dynamics. Archetyped software overcomes all the limitations found with Ada generics. Archetyped part elements, combined with a formal grammar, will provide context-sensitive expansion of specifications into compilable Ada source code.

It is the conclusion of this paper that absolute certification is a desirable but extremely difficult to achieve goal. Partial certification is a more realistic goal and is attainable with existing technologies.

Moreover, in order to use parts "as is", they must be kept small and uncomplicated, otherwise the process of certification becomes very complex. A methodology, such as archetyping, combined with the proper tools, can make parts adaptable, reduce complexity, and allow for reuse of larger bodies of code.

The concepts described in this paper reflect research being performed at General Dynamics Data Systems Division, San Diego, California.

(1) Przybylinski, S. "Archetyping- A Knowledge-Based Reuse Paradigm" April, 1986