The Form of Reusable Ada Components for Concurrent Use

Jeffrey R. Carter
Senior Engineer, Software
Martin Marietta Astronautics Group
P. O. Box 179
Denver, CO 80201

Abstract

Abstract data types are often chosen to implement reusable Ada components for sequential use. Those who have attempted to expand from sequential to concurrent considerations have frequently continued to use abstract data types for their implementation. This approach is shown to violate software engineering principles. The use of abstract state machines for concurrent components is shown to resolve these problems.

Introduction

In my work with the MMAIM notation [1, 2, 3] I have found that it encourages the use of abstract state machines over abstract data types for components for concurrent use, although it shows no such preference under sequential conditions. Most discussions of Ada components emphasize the use of abstract data types. For example, Genillard *et al.* [4] and Booch [5] both implement their components as abstract data types. Genillard *et al.*'s major changes from Booch are that they consistently provide an "assign" procedure for abstract data types implemented by limited private types, and they provide a form of their components which accept a limited private type, with an "assign" procedure, as the element type of the component. This allows them to use the abstract data type provided by one component as the element type of another component to build structures of structures.

Abstract data types are an excellent form for reusable Ada components for sequential use. They have no disadvantages when compared to abstract state machines, and have the advantage that, if properly designed, they allow building structures of structures.

Booch implements most of his abstract data types as limited private types, but most of his components only accept private types as the element type of the component. Thus, one cannot build structures of structures from his components. However, he has provided forms for concurrent use. His concurrent components are implemented as abstract data types with a task component. From a software engineering point of view, there are some problems with this approach.

Problems with Concurrent Abstract Data Types

Booch uses a binary semaphore task to handle concurrent access to his abstract data types. The semaphore is an obsolete construct for controlling concurrency which has been replaced in modern languages by higher-level constructs such as Ada tasks and rendezvous. The use of a binary semaphore is analogous to using GOTO's: A programmer may, accidentally or maliciously, violate the higher-level concept when using the low-level constructs, but not when using the higher-level constructs.

Booch exports the binary semaphore in his guarded forms (see, for example, his guarded ring [5, p. 193]), which allows a user of the component, accidentally or maliciously, to violate the integrity of the abstraction. Nothing prevents a user from operating on the structure without first seizing it, nor is there anything to prevent a user which has seized the structure from never releasing it. It is not good software engineering practice to allow the user to violate an abstraction's integrity. It is fairly simple to provide a controlled form of components which provide the same functionality as Booch's guarded form without allowing the user to violate the abstraction's integrity. For example:

```
generic -- simpleminded controlled stack
   type element is limited private;
   with procedure assign (to : in out element; from : in element);
   no_access_interval_for_automatic_release : duration := 1.0;
package simpleminded_controlled_stack is
   type stack  is limited private;
   type key_id is limited private;
   procedure request (access_to : in out stack; key : out key_id);
   invalid_key : exception;
```

```
      procedure assign
         (to : in out stack; from : in stack; to_key : in key_id;
         from_key : in key_id); -- raise invalid_key
      procedure put
         (onto : in out stack; item : in element; key : in key_id);
         -- raise invalid_key
      procedure get
         (from : in out stack; item : out element; key : in key_id);
         -- raise invalid_key
      procedure release (use_of : in out stack; key : in key_id);
         -- raise invalid_key
   private -- simpleminded_controlled_stack
      ...
   end simpleminded_controlled_stack;
```

In this example, failure to first request access to the stack results in operating on the stack with an invalid key, causing the exception invalid_key to be raised. Once a stack has been requested, if a period as long as the generic parameter no_access_interval_for_automatic_release goes by without an operation being performed, a release is automatically performed, allowing access to another user and making the first user's key invalid.

Using shared variables violates the software engineering principles of information hiding and locality. Using shared variables with concurrent access is even worse software engineering practice, yet using abstract data types to implement the concurrent forms of components (as in this example) requires that the user's tasks share variables. It is impossible to use a concurrent component implemented as an abstract data type without having tasks which share variables.

The Abstract State Machine Alternative

Luckily, there is a concurrent form which does not violate good software engineering practice. To achieve this, use an abstract state machine instead of an abstract data type to implement the component. We cannot build structures of structures using only abstract state machines, but I believe that this is rarely needed. While the final component must function correctly under concurrent access and be an abstract state machine, the elements of this component can usually be for sequential use only and be implemented by abstract data types. This allows us to build structures of structures and still use an abstract state machine for concurrent components.

Concurrent abstract state machines fit nicely into graphical software development notations. Concurrent queue abstract state machines are an essential part of the JSD notation [6], and concurrent abstract state machines fit more easily than concurrent abstract data types into the Mascot-3 [7] and MMAIM notations. In solving the RDAS problem I used a concurrent queue abstract state machine as the incoming character buffer [2, Figure 6]. Booch diagrams [8] and Buhr diagrams [9], while more closely tied to Ada structures, also handle concurrent abstract state machines more easily than concurrent abstract data types.

One need only to try using one of these notations with both a concurrent abstract data type and a concurrent abstract state machine to see why the notations favor abstract state machines: Abstract data types complicate the design of concurrent software. A simple example demonstrates this. Concurrent queues are often used to allow asynchronous communication between independent threads of control. Using a concurrent abstract data type queue and the MMAIM notation produces Figure 1. T1 and T2 are concurrent entities which communicate asynchronously using the shared variable E.Q which is a concurrent queue of the abstract data type D_Q.Q_TYPE. The elements of this queue are never accessed concurrently.

Using a concurrent abstract state machine queue produces Figure 2. T1 and T2 are concurrent entities which communicate asynchronously using a concurrent queue abstract state machine, D_Q. Figure 2 is simpler and clearer than Figure 1. Figure 1 represents a more complicated design than Figure 2.

This difference would not occur for a sequential system. Whether one uses an abstract data type or an abstract state machine, the result would show one entity (T1) calling one reusable component (D_Q).

The more complex design resulting from using abstract data types makes a system harder to understand and modify, increasing the cost of operating the system. Although not necessarily the case, a design which is hard to understand is often hard to create and implement. This implies that systems designed using abstract data types may be more expensive to develop than systems designed using abstract state machines.

Summary

Abstract data types are an excellent form for implementing reusable Ada components for sequential use. They have the benefit that, when properly designed, they allow building structures of structures. However, for concurrent reusable Ada components, using abstract data types violates software engineering principles and may lead to more expensive software which is difficult to understand and modify. Using abstract state machines avoids these problems and fits nicely into modern, graphical software development methods. Under concurrent conditions, an abstract state machine is the preferred form for reusable Ada components.

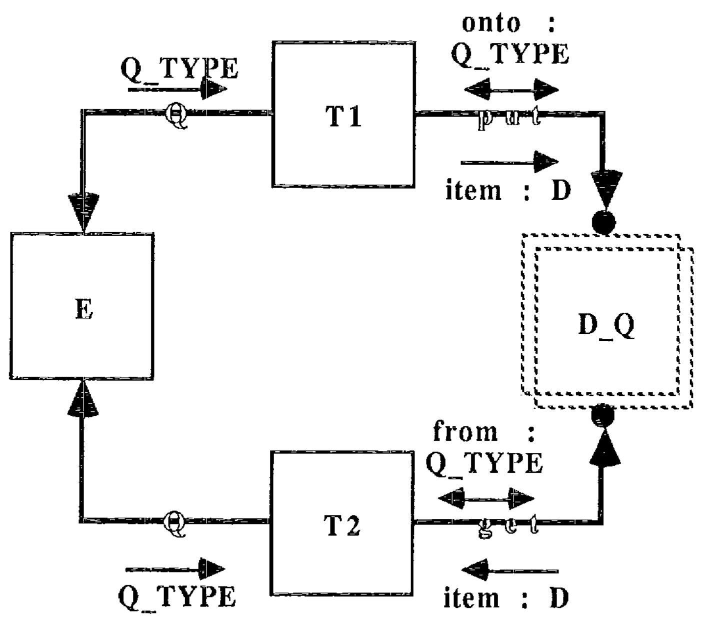

Figure 1
T1 and T2 communicate via E.Q

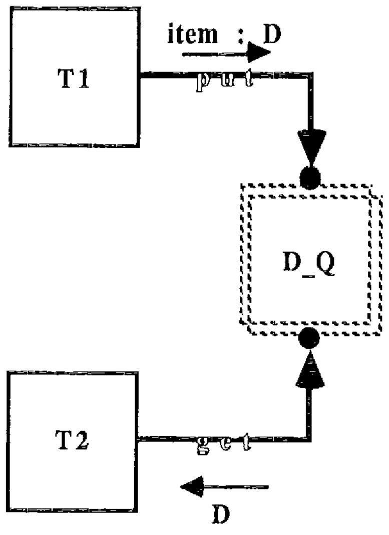

Figure 2
T1 and T2 communicate via D_Q

References

1. Carter, J., "MMAIM: A Software Development Method for Ada; Part I– Description," *Ada Letters*, 1988 May/Jun
2. Carter, J., "MMAIM: A Software Development Method for Ada; Part II– Example," *Ada Letters*, 1988 Sep/Oct
3. Carter, J., "Reducing Software Development Costs with Ada," *Proceedings of the Seventh Annual National Conference on Ada Technology*, ANCOST, Inc., 1989
4. Genillard, C., N. Ebel, and A. Strohmeier, "Rationale for the Design of Reusable Abstract Data Types Implemented in Ada," *Ada Letters*, 1989 Mar/Apr
5. Booch, G., *Software Components with Ada*, Benjamin/Cummings, 1987
6. Cameron, J., "An Overview of JSD," *IEEE Transactions on Software Engineering*, 1986 Feb
7. Bate, G., *The Official Handbook of Mascot, Version 3.1, Issue 1*, Royal Signals and Radar Establishment, 1987
8. Booch, G., *Software Engineering with Ada, Second Edition*, Benjamin/Cummings, 1987
9. Buhr, R., *System Design with Ada*, Prentice-Hall, 1984