SOME DESIGN CONSTRAINTS REQUIRED FOR THE USE OF GENERIC SOFTWARE IN EMBEDDED SYSTEMS: PACKAGES WHICH MANAGE ABSTRACT DYNAMIC STRUCTURES WITHOUT THE NEED FOR GARBAGE COLLECTION

Charles S. Johnson

ABSTRACT

The embedded systems running real-time applications, for which Ada was designed, require their own mechanisms for the management of dynamically allocated storage. There is a need for packages which manage their own internal structures to control their deallocation as well, due to the performance implications of garbage collection by the KAPSE. This places a new requirement upon the design of generic packages which manage generically structured private types built-up from application-defined input types. These kinds of generic packages should figure greatly in the development of lower-level software such as operating systems, schedulers, controllers and device drivers; and will manage structures such as queues, stacks, link-lists, files, and binary/multary (hierarchical) trees. Generic structures like these will have to be carefully controlled to prevent inadvertent de-designation of dynamic elements, which is implicit in the assignment operation. A study is made of the use of the limited private type, in solving the problems of controlling the accumulation of anonymous, detached objects in running systems. The use of deallocator procedures for run-down of application-defined input types during deallocation operations is also discussed.

INTRODUCTION

Reusability is crucial to programs developed for Integration and Test (I & T) applications. The Ada language was specifically developed for use on embedded systems where most of the real-time applications work is performed. The creation of a software support environment for real-time work must first deal with the selection of a design approach which maximizes the reusability of Ada software components. The issue of Ada reusability does not just address problems of portability across machines and between projects, but also reusability within one project, and for one machine. One property of generic abstraction is the containment of a solution for a system- and application-dependent problem. Once having been solved generically, that solution is available for reliable reuse by all the applications of the system.

BRIEF BACKGROUND

Kennedy Space Center/ Engineering Development/ Digital Electronics Engineering Division is in the process of prototyping distributed systems supporting I & T applications, particularly the Space Station Operations Language (SSOL)

System, which is the I & T subset of the User Interface Language (UIL) for the Space Station. The discussions in this paper were developed from the results of systems designed and developed in Ada to demonstrate the feasibility of developing reusable software specifically targeted for real-time embedded applications. The Ada environment used was that of VAX Ada under VAX/VMS.

USE OF ADA IN EMBEDDED SYSTEMS

The implementation of the Ada KAPSE for a computer system can be performed in one of two ways. The KAPSE can be layered over an existing operating system, using it's services and saddled with it's limitations. The KAPSE can also be directly layered onto the computer hardware, and act as a limited operating system. Ancillary operating system services will then need to be supplied by Ada applications. For most embedded systems the latter alternative will hold, for both developmental and performance reasons. Developmentally, it is harder to re-host both the operating system and the KAPSE to new computer hardware, than it is to re-host the KAPSE alone. Also, for applications developed on a layered KAPSE, performance will suffer as requests for system services have to be processed at two levels. The organization and system approach for the two levels of support, since they were not designed specifically to be integrated, will almost certainly be mismatched in many ways.

For systems with a native KAPSE, the optional features of the Ada language (some pragmas, services) will be slow in appearing, or may be seen to be negative in effect. The system garbage collection feature in the KAPSE will be one of those features that won't appear initially. When it does appear, in many implementations, it's use will be precluded in real-time systems. [1]

The garbage collection feature of the KAPSE tracks, and deallocates anonymous objects in the Ada system, thereby freeing the system resources that they use.

Anonymous objects are previously-designated objects of a type associated with an access type (pointer type). A designated object is created by an allocator, which associates it with an access object (pointer object), which then, of course, designates it. Designated objects are implicitly declared by that allocation as objects of the designated subtype (subtype of object pointed to) of the access type, and are compatible with all objects declared of the designated base type (original type referenced in the access type definition).

Designated objects become anonymous objects by three means, all have to do with assignment:

1. The access object designating the object is assigned to the value of another access object of the same type.

2. The access object designating the object is assigned to a new value by an allocator.

3. The access object designating the object is assigned to the value "null".

Unless the previously-designated object was designated by more than one access object, after access object reassignment it becomes an anonymous object.

The use of access types is necessary if a system is to be flexible, and capable of creating objects in response to needs that cannot be specified until the need arises. Release of the system resources used by objects of designated subtypes, is essential in that flexibility (or static types rather than dynamic types could have been initially specified).

In layered systems built on general-purpose operating systems, the tracking down and subsequent deallocation of the resources consumed by these anonymous objects (the garbage-collection process) will be a built-in feature. In VAX-VMS the KAPSE performs this service. In AT&T Ada for the AT&T UNIX System V (Release 3), this service is implicit in the system, because all Ada objects are created on the system heap, which is managed by the system. In both cases, there is an ever-present background process, performing rundown of dynamic objects declared in the system. The performance detriment due to this background process is unpredictable, both for when it occurs (it is concurrent and unsynchronized with the applications) and for the system resources it consumes.

It is noted here that access types can be both data and task types. The problem of garbage-collection exists for both task and data types. In this paper, only the data type problem will be discussed.

There is no requirement in the Ada Reference Manual (ARM) [2] for the garbage collection feature to be implemented in the KAPSE. For many embedded systems running real-time applications, it will be required that the garbage-collection feature, if present in the KAPSE, retain the capability of being turned off. The presence of unpredictable resource consumption is contradictory to the principals of real-time computing, in particular, the response to external interrupts in a timely and reliable manner.

This poses a new problem. Without garbage-collection, the only time that anonymous objects are collected by the system (deallocated), is upon the expiration of the scope of the application which contains the definition of the access type. For anything other than restrictive use of the access type, this will usually be a package specified at the highest scope in the program. This scope, by not expiring, implies that normal collection will never occur (without garbage-collection).

For programs running on embedded systems, this means dynamic objects will continuously be converted into anonymous objects, consuming more and more system resources, until the program aborts when the system resources are exhausted. This self-destructive behavior may not be noticed during verification

or validation, if the process of creating anonymous objects is sufficiently slow. Indeed, well-written processes that are conservative in their exhaustion of system resources may live long before the limitations are breached.

These programs must, then, control their own storage allocation and deallocation. A pragma for declaring the storage management for an object as being controlled by the application (pragma CONTROLLED), and a generic package for deallocating controlled objects (UNCHECKED_DEALLOCATION) will be available for embedded systems development. The problem is that the implementation of these features must be standardized in the development of the application system, for there to be any assurance that anonymous objects will not collect.

A design philosophy encouraging abstraction would tend to drive the Ada source code using these features into the hidden scope of a package. This would create, in the system, an assortment of packages which define, declare and manage private access types, while retaining complete control of the allocation and deallocation of objects designated by those types. The control of the storage allocation in these packages would need to be implemented in an efficient way, such that the use of the package types would be flexible and easy (to encourage package use). A requirement of these packages, stemming from real-time considerations, would be that the behavior of systems using these packages should differ from that of systems using garbage-collection. The overhead incurred by the deallocation of storage should occur in predictable amounts, and in synchrony with, or under the control of the operation that incurs the overhead.

A design philosophy encouraging maximum reusability of software for the system, would tend to drive those packages, where possible, into a smaller family of generic packages using generic formal parameters which determine the differences between instantiations. Maximum reusability of these generic packages could be accomplished by the use of generic formal parameters matching the widest variety of input types, and by declaring internally controlled dynamic types which match the widest variety of applications (flexibility of use).

GENERICALLY STRUCTURED ABSTRACT TYPES

At some point in most Ada textbooks, a generic package is described that maintains a generically structured abstract type. The type is declared inside the package, and contains a component type within it which is defined from a generic formal type parameter (an application defined type contained within a generic structure). The example given is typically for a generic stack, list or queue, and the generically structured object may be hidden within the package, or declared as private type, or just as a type.

The important point of these textbock examples is the demonstration that the procedures for managing even very complex structures such as lists, queues, binary trees, multary (hierarchical) trees and files can be made general and separated from the procedures for managing the objects that they

contain. And, of course, that Ada supports the separation of these management procedures in a slick and easy-to-use manner.

If the design constraints on the system (storage control) can be embedded into the packages managing generic structures composed of application-defined types, many possibilities open up. The creation of what could be very complex systems such as operating systems containing schedulers, controllers and drivers becomes much simpler. These kinds of programs can be based on the use of just a few simple types of structure.

In an example, if a generic structure such as an index were managed in a storage controlled way, many system structures and much system processing could be based upon it. An index is a list of elements of one type (can be composite), ordered by elements of a second type, the index key. Many sample applications are possible. Logons could be controlled by a list of user names versus passwords, ID's, priorities, etc. Batch printing could be performed using a priority ordered list of print files. A disk directory could be held as a list of files ordered by name, or lists of lists. Batch scheduling of tasks could be ordered by priority or timestamp. More pertinent to I & T applications, a list of logical designators for the control of hardware on a Test System could order the blocks which contain their logical-to-physical access information. In this case a hierarchically ordered list of designators versus access blocks would probably be more useful.

The focal point of the impact of this technology is on the reuse of software components within a project. The system-dependent functioning buried in the body of packages, will not be nearly as portable between machines and areas of application as it is reusable within a project. Some external software will be incorporated, of course, like it is today: DBMS, graphics support, user interface packages, communications support. These kinds of packages will be available where there are broad areas of commonality of function, and where system-dependent features can be profitably developed in packages by vendors.

Standardization by the use of generically managed structures makes possible the idea of technology insertion directly into the applications of a system. If a system- or application-dependent problem is solved one time, in a flexible and reusable manner, the developer can beat that solution to death, reusing it over and over.

Maintenance of reusable software enhances the system effectiveness. That reusable solution can be tuned at a minimum number of locations in the system, and re-inserted into the applications. If a better hashing function is found for the key of our index example, for instance, a widespread increase in performance will result.

DESIGN GOALS AND CONSTRAINTS

The design of packages managing generically structured abstract objects must begin with the establishment of goals and

constraints. The goals and some of the constraints are independent of the problem of embedded systems. [3]:

1. Package-managed generic objects that are declared in the application software should, where possible, be defined as abstract types, that is, made private.

2. Maximize the generality of the package. This comes from the use of formal generic parameters, particularly for types, that match the widest variety of application input types (type private instead of digits <>, for example).

3. Maximize the usability of the application interface to the package. Extend, as far as possible into the application domain, access to the structures managed in the package, without violating the integrity of the internals, or the independence of the application from the generic software component (generality).

4. Maximize the completeness of the application interface to the package. Give the application developer all the operations required to access and manipulate the internal structures, in a package-controlled manner.

5. Support, if possible, multiple objects with the same package. This limits the need to re-instantiate the package several times within the same scope, for processing of multiple objects.

6. Design for flexibility: a single tool, suited to a wide range of applications, is more likely to be remembered, and used by developers.

7. Cover the infrequent failure modes. Most failures of algorithms and processing logic in programs occur at the extremes of their domain of applicability. Testing should cover the ends of ranges and the infrequent states of the application. If the software component is reusable, it will be used in a wider range of applications, and the infrequent failure modes will occur more frequently.

Some of the constraints on the design of packages managing generically-structured abstract objects stem from requirements generated by the use of Ada on embedded systems, and are therefore application-dependent:

8. The package operations must control and deallocate any internally allocated dynamic storage.

9. The package must, by it's implementation, disallow any inadvertent de-designation of package managed dynamic

structures or elements. The application must be prevented from creating anonymous objects.

10. The overhead involved in the processing of package operations must be predictable and controllable by the application (in contrast to the garbage collection of anonymous objects by the KAPSE).

SELECTION OF DESIGN APPROACH

The index package, which was described above as a list of elements ordered by another set of associated index key elements, will be used as an example for the selection of design approach. The index structure itself should be some kind of private type. Functions for index lookup by key item, element add/delete, and for stepping through the index sequentially should provide a useful set of operations for index manipulation. The INDEX type itself should be defined in the package specification, not hidden, so that it can be declared as an object in the package scope.

The importance of having the index object in the scope of the application is in the flexibility of use of the object at the application level. The developer should be capable of passing the object as a parameter to subprograms developed at the higher level. If the object of type INDEX is hidden, this flexibility is not there.

This generates a conflict with the application-specific constraint about allowing the application to inadvertently generate anonymous objects. If the object of type INDEX is declared in the user scope, any kind of assignment operation to it will create an anonymous INDEX object.

USE OF THE LIMITED PRIVATE TYPE

The definition of the INDEX type as limited private prevents reassignment of it's value in any operation. It cannot be reassigned in the deepest level of any procedure (Ada), or generic software component that knows of it's typing. This allows the access object to be declared in the user scope, and used as a parameter, without any chance of creating anonymous objects from reassignment (unless the package itself does).

The removal of needed functionality by the definition of the type as limited private, creates a need for the definition of analogous functions: assignability, comparability, nullability.

The assignment function which has been removed cannot be replaced exactly. If the application is given the ability to assign the same value to INDEX objects, even controlling the creation of anonymous objects during reassignment won't help. Having two INDEX objects of the same value implies that the package cannot explicitly deallocate either INDEX designated object, without creating an erroneous circumstance (an INDEX object designating a deallocated object). This cannot be allowed. Therefore assignment (call it ASSIGN the ":="

operator cannot be overloaded) will first clear the access object value by deallocating the current designated object, and then copy the object designated for assignment, element for element, until two copies exist.

If the need for mutual designation by the same INDEX object was a requirement, creation of anonymous objects could be controlled by the installation in the structure of the INDEX designated object of a semaphore-type variable, which would provide concurrent access to the structure along with the protection by mutual exclusion. This would allow the package to keep a count of the number of INDEX objects accessing the structure of the index, with the capability to deallocate the INDEX designated object upon the reassignment of the last INDEX object designating it.

The compare function, "=", can be overloaded for limited private types, and should be defined to compare the elements designated by the two objects of type INDEX, one for one, to establish equality. It should be noted here, that the application itself could define "=", if the capability of stepping through the INDEX elements one by one, and retrieval functions for each element are provided.

The re-initialization of the INDEX object ("null" assignment) is replaced by a DELETE function which deallocates the designated object (the entire structure).

APPLICATION DEFINED DEALLOCATOR PROCEDURES

There is one last potential for the inadvertent creation of anonymous objects by the package itself. The package allocates a node when it adds an element to the INDEX designated object, and it deallocates a node when a delete of an element occurs. However, if the type that was passed as the formal generic parameter for the key type or the element type is itself an access type, deallocation of the node will create anonymous objects that were previously designated by access objects of the application-defined input types.

The solution for this problem depends upon the developer. For every application-defined component type which is passed into the generic package as a generic formal parameter to be incorporated into a generically-structured storage-managed type, there must be an accompanying generic formal parameter indicating a procedure which deallocates any objects designated by an object of the application defined component type. This allows the generic package to invoke that procedure for the components of the structure, so that the subsequent component deallocation will not create any anonymous objects.

For application-defined types that are not or do not contain access objects, the deallocator procedure passed would simply provide a null return, and do nothing.

To repeat this rather complicated rule in other words, there is a need for every generic formal parameter of an application-defined type for a structural component, to have an accompanying deallocator procedure, not for the type itself, but for designated objects of that type, and designated objects of

those designated objects, and so on. If the developer wishes to incorporate structures within structures, the price of this complexity must be paid.

INCREMENTAL DELETE FEATURE

It is not reasonable to assume that the size of the structure being managed by the generic is known before the application is coded, or else the developer might have chosen a static rather than a dynamic type. The processing overhead incurred from the deletion of an entire structure or one part of a structure is then also not predictable. This can put the real-time performance of the package operations back to square one.

If a real-time application performs a delete operation, the return from the subprogram must be made within application defined time-constraints for the package to be useful. In an example indicating the problem, a real-time application, while in between accepting interrupt entries from a hardware device (a time-critical operation, for hardware interrupts are not queued), attempts to initialize the access object designating a structure, during the time window that is known to exist between interrupts. During initialization of the structure it is necessary, of course, to run down the entire structure, deallocating each component of the current structure exhaustively, until the access object can be initialized. Unfortunately, during the time that the subprogram took control away from the real-time application, several interrupts were overwritten, and critical data was lost.

The solution to this problem is to supply an incremental delete function. The overhead incurred from the delete and subsequent deallocation of a single element is knowable. An incremental delete operation can then be defined, such that upon input of the logical parameter indicating how much of the structure to remove, and a physical parameter indicating the number of elements to remove for each successive invocation, the structure will be whittled away incrementally. The order of deletion/deallocation should be such that a reference always exists to the remaining increments of the section of the structure that are to be removed (for example, delete a tree from the leaves in toward the root).

CONCLUSION

It is concluded, by our studies, that it is feasible to create families of highly reusable generic software components, specifically tailored to support kinds of applications. These generic packages can maximize the reusability of software developed within and for a particular project. At the same time they can address the performance requirements of software developed for embedded systems running real-time applications. These requirements stipulate that such software be responsive and controllable in terms of direct processing overhead, and incur little or no background processing overhead of an

unpredictable nature (in contrast to the garbage collection of anonymous objects by the KAPSE).

ACKNOWLEDGEMENT

I gratefully acknowledge the support given by the Kennedy Space Center/ Engineering Development/ Systems Integration Branch in supplying the computer facilities for the feasibility studies that provided the basis of this work. I also thank my wife, Bronwen Chandler, for her support.

REFERENCES

1. Burns, A. 1985. Concurrent Programming In Ada. Cambridge, Great Britain: Cambridge University Press.

2. United States Department of Defense. February 17, 1983. Reference Manual for the ADA Programming Language. ANSI/MIL-STD-1815A-1983. New York, New York: Springer-Verlag.

3. Johnson, C., 1986. "Some Design Constraints Required for the Assembly of Software Components: The Incorporation of Atomic Abstract Types into Generically Structured Abstract Types", Proceedings of the First International Conference On Ada* Programming Language Applications For The NASA Space Station, E.1.1.