The Objectization Problem - What It Is And What Can be Done About It

Aloysius K. Mok†
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

1. The Narrow Interface of Middleware

The packaging of both data and procedures into autonomous units is one of the key insights exploited in object-oriented programming. By projecting the system state space into independent object spaces and by limiting the effects of one object's actions on another through an object's methods, implementation changes in one object will not be visible to other objects. This clean delineation into object spaces is also amenable to the application of compositional proof techniques in formal methods. That is the theory.

We shall show that for real-time embedded systems where real-time performance and reliability are as important concerns as logical correctness, the decomposition of a system into objects may require a *design commitment* to satisfy multiple and often conflicting criteria. Such design commitments come about because the choice of objects determines the granularity of the architectural unit for modularization, not only for specifying a software component but also for resource allocation and security/fault-tolerance enforcement. To achieve modularity, the granularity of the architectural unit for modularization may vary for different types of correctness concerns such as real-time performance and security/fault-tolerance. Unless we can provide the technology to "wrap" legacy software in multiple ways according to the granularity required, we may not be able to realize the promised benefits of middleware, inasmuch as the interaction between new and legacy software may not be that easily contained to a narrow interface. Firstly, let us understand the problem.

2. The Objectization Problem

In order to transform an application requirements into operational software, the computational requirements of the application must eventually be mapped into a set of architectural units. The traditional architectural units are those supplied by the operating system: processes and files. In the era of object-oriented programming, the architectural unit is the object which comprises of a set of methods and a non-shared data space on which the methods of the object operate. We call the mapping of application requirements into objects the **objectization problem**.

In the context of the objectization, the *object* architectural unit can be seen to serve at least three different purposes which might conflict with one another. Specifically, an

† Supported by a research grant from the Office of Naval Research under ONR contract number N00014-94-1-0582

object often serves as an architectural unit for: (1) Resource allocation, e.g., the scheduling unit for resource allocation, (2) Integrity management, e.g., a guardian of shared data, and (3) Task requirement capture, e.g., to track an airplane in an air traffic control system. These three different roles of the object may require the application requirements to be broken up in different granularities and thereby impose an unnecessarily difficult tradeoff in objectization. Let us examine these conflicts in some detail.

2.1 Evaluation of Objectization Schemes

We shall limit our discussion to three types of conflicts corresponding to the different combinations (pairs) of the three roles of the object architectural unit. We shall illustrate each of these conflicts with an example application. For each example, we shall give two different objectization scheme each of which is a good decomposition with respect to one role of the object but a bad decomposition with respect to another role of the object. The evaluation criteria we shall use to measure the "goodness" of an objectization scheme are: (1) For resource allocation, a good objectization scheme should minimize the usage of resources. (2) For integrity management, a good scheme should minimize the system impact caused by the failures of some objects. (3) For task requirement capture, a good scheme should minimize the efforts required to make incremental modifications to the program and to verify its correctness. Our evaluation criteria are largely qualitative; we believe that a quantitative approach will not substantively change our conclusions and may instead obscure the general nature of our observations.

2.2 The Resource Allocation vs. Task Requirement Capture Conflict

Consider a real-time process control application which requires computation to be carried out in three algorithms X, Y and U. The algorithms X and Y read as input respectively the external signals x and y, and the algorithm U computes an output u from the most recent values of the outputs of X and Y. The real-time performance requirements of the application consist of two tasks: (1) Sample x and update u every 4 milliseconds. (2) Sample y and update u every 8 milliseconds. We now consider two alternative objectization schemes.

2.2.1 Scheme 1:

The first objectization scheme creates an object for each of the two tasks. The first object is to be scheduled every 4 milliseconds and executes X and then U every time it is scheduled. The second object is to be scheduled every 8 milliseconds and executes Y and then U. If each of X, Y and U takes 1 millisecond to execute, it is easy to check that this system of two objects can be scheduled on a single processor. Notice that under this mapping, an object is simply the image of a task in the application requirement. Additional tasks (e.g., there is another input signal to be processed) can be incorporated as objects with no modification to the objects already in the system.

2.2.2 Scheme 2:

An alternative objectization scheme creates a single object which interleaves the execution of the two tasks. To emphasize the potential advantage of this mapping, assume that

Y takes 4 instead of 1 millisecond to execute. The single object in this objectization scheme is to be scheduled every 4 milliseconds (a minor cycle) and executes X and U twice and Y once every 8 milliseconds (a major cycle). In the first minor cycle, X is executed, followed by the first half of Y, and then U is executed using the output at Y from the previous major cycle (Y has not finished execution at this point). In the second minor cycle, X is executed, followed by the second half of Y creating a new output, and then U is executed using the new output of Y.

The reader can easily verify that scheme 1 will now be unable to meet the periodic timing constraints, but these timing constraints can be met by the second scheme. Even though objectization scheme 2 is more efficient in terms of resource utilization, it is considerably more difficult to understand and to modify than scheme 1; the correspondence between objects and task requirements is no longer maintained. If we know that the availability of CPU resources is not critical, then scheme 1 is more preferable. Scheme 1, however, embodies a design commitment which is supposed to hold throughout the entire life cycle of the system, an unrealistic assumption at best.

2.3 The Resource Allocation vs. Integrity Management Conflict

Consider the well-known producer-consumer paradigm in operating system design where a producer outputs to a buffer from which consumers receive the commodity.

2.3.1 Scheme 1:

The producer and consumers are objects which gain access to the buffer by invoking lock and unlock method on a semaphore object. This objectization scheme is efficient in terms of CPU usage, but is not robust For example, if one of the consumers fails inside the critical region and either hangs up the system or traps out by invoking an exception handler, then the buffer will be left in an inconsistent state and will not be accessible to the other customers. Thus the application may grind to a halt just because of a single customer object fails at the wrong time.

2.3.2 Scheme 2:

An object is created as the *monitor* for the buffer. Both the producer and consumers must interact with the monitor via an established protocol to gain access to the buffer. This objectization scheme has better fault-tolerance properties than scheme 1 in that if a customer object fails inside the critical region, the monitor object can release the buffer when it detects, say via a timeout mechanism, that the customer object accessing the buffer has failed. However, this scheme is less efficient from the performance point of view since several context switches are incurred by the producer, the monitor and the customer for every buffer transaction.

2.4 The Integrity Management vs. Task Requirement Capture Conflict

Consider a distributed system where transactions are to be performed between users located at two sites in the network. There are many types of transactions which require different resources and they can be invoked via a multi-layer communication protocol. More than one pair of users (tasks) may be communicating with one another between the two sites. Each layer of the communication protocol makes use of a number of resources (virtual and physical) which are shared among all users.

2.4.1 Scheme 1:

We create one object for each layer at each site in the network. This object establishes a single virtual channel between the two sites and multiplexes communication between all communicating user-pairs at that layer. This object acts as a guardian for the resources at the layer and performs atomicity/recovery management for that layer. When two users want to engage in a transaction, they interact exclusively through the guardian object at the top layer of the communication protocol which catalogs all transactions. This scheme has good fault tolerance properties since it is easier to contain partial failure propagation to within a layer and the recovery overhead is likely to be smaller.

2.4.2 Scheme 2:

A pair of objects are dynamically created for every pair of communicating users, one at each site, i.e., each transaction is modelled by a pair of objects. This object-pair establishes an individual virtual channel between the two sites and performs the transaction required for the two users. Since more than one pair of users may be interacting at a time and object-pairs may execute at every layer, locks or other distributed synchronization mechanisms are needed to maintain the integrity of the logical and physical resources shared by all object-pairs (transactions).

Scheme 2 is more flexible than scheme 1 since we can readily add new types of transactions which might otherwise require modifications to many of the objects implementing the different layers of the communication protocol. Also, some transactions may not need to go through every layer in the protocol and thus the corresponding object-pairs can be tailored to perform the required task more directly and efficiently. The problem with scheme 2 is of course its lack of robustness. Since every object-pair in effect executes a fragment of the layered communication protocol, it is considerably more difficult to ensure that the failure of one transaction will not cause uncontained damage to others.

3. Lessons Learned

The lesson from the above discussion on granularity conflicts can be summarized as follows. In decomposing application requirements into objects, there may not be a single best objectization scheme which can serve orthogonal design concerns. What is good modularity with respect to one concern in system design may not be good for another.

In a gradual replacement of components strategy for transitioning formal methods and other advanced techniques into current designs, it is not enough to consider modularization from the functional dependency point of view. Middleware or wrapper technology built with this view may end up causing accidental performance bottlenecks and/or failure modes which should not be tolerated.

4. What Can Be Done

It would be best if different objectization schemes can be used for meeting different design concerns. This of course introduces the problem of consistency maintenance among different objectization schemes if they are to describe different aspects of the same system. Another way to look at this issue is to consider different *views* of an objectization scheme. Suppose we can project the real-time behavior view from an objectization scheme of a system; evolving functional requirements may cause a change in the objectization scheme which in turn induces a change in the real-time behavior. Then the consistency of the two views: functional and real-time behavior can be maintained by making sure that the induced change in the real-time behavior does not violate the real-time performance specification of the system. This requires broadening the concept of an interface for objects to go beyond method invocation templates to include other properties such as real-time behavioral specification. Research is needed to support compatibility checks for non-functional properties.

We have been working on an event-based annotation scheme and a logic [JAH & MOK 86] for capturing assumptions made about the execution environment. We are currently developing tools to automate the feasibility analysis for composing real-time performance requirements of complex time-critical systems. Some progress has been made, and the details of the application of our tool MSP.RTL to the AIMS system of the Boeing 777 aircraft can be found in [MOK et al 96]. We are planning to further explore this problem in relation to Locke's [LOCKE 97] four canonical architectures for real-time systems.

Bibliography

[JAH & MOK 86] "Safety Analysis of Timing Properties in Real-Time Systems," F. Jahanian and A.K. Mok, *IEEE Transactions on Software Engineering, Vol. SE-12, No.9, September 1986.*

[MOK et al 96] "The MSP.RTL Real-Time Scheduler Synthesis Tool," A.K. Mok, D.C. Tsou and R.C.M. de Rooij, *Proceedings of the 17th IEEE Real-Time Systems Symposium, Washington D.C., December 1996, pp. 118-128.*

[LOCKE 97] Private communication, C. Douglas Locke, Chief Scientist, Lockheed Martin Corporation Software & Systems Resource Center, 1997.