OOPSLA '93
Object-Oriented Reflection and Metalevel Architectures Workshop Submission

John W. Gilbert
Objective Software
1985 W. Henderson Rd., Suite 525
Columbus, Ohio 43220
email: 72530.141@compuserve.com

In the construction and management of complex systems in the areas of scientific, real-time, and multimedia applications, the ability of software to reason about itself is essential. These systems often require distributed processing in hazardous locations or over large geographic areas or to support sensitive missions, where reliability, resiliency, and adaptability are essential. The science of computational reflection has in recent years allowed a better understanding of how to structure software and particularly object-oriented software to support reflective behaviors in the management of computational resources, software configuration, fault tolerance, and reusable componentry.

The challenge today is to define and design reusable architectures for the support of reflective systems where the mechanisms are not tied to a particular programming language or environment. The latter is difficult in that there must be some common materials from which to build systems. So, the trick is to create a reusable reflective architecture which can be implemented in any language.

In object-oriented systems, languages often (wrongly) drive the design and architecture, but complex systems, which cannot assume a uniform implementation environment, must structure objects independently of language. Large efforts such as CORBA and ANSA are creating such systems, but these systems and their object models are much to heavy weight and not usable for systems which require high degrees of efficiency or timeliness. As a result many organizations are building their own object models to support their specific but often diverse requirements. In these systems what are the reflective requirements and how can reflection be built in?

Real-time systems serve as an excellent example of the type of complex system which is being built (in some instances from the ground up) using object-oriented and reflective architectures. These systems are characterized as those which must manage:

* computational constraints, such as
    - time
    - priority
    - consistent data rates

* adaptability, such as:
    - dynamic configuration
    - fault tolerance
    - migration

Real-time means accomplishing tasks within the time required. This does not always translate into fast. For many industrial process control systems fast processing is important, but equally important is deterministic behavior. Appropriate control outputs must be delivered on-time to ensure safe plant operation.

In some circumstances, certain actions are more important than others. A safety condition in a boiler system must be handled at a high priority. Priority schemes are utilized to manage these needs.

Other types of real-time systems are concerned with regular delivery of data, at a guaranteed rate. Multimedia systems require a constant rate of data to avoid noticeable dropouts in sound or action.

To support these time oriented constraints objects or groups of objects which service these requests must incorporate time and priority into their interactions. This is reflective information which in order to satisfy the constraint must be propagated along with the computation. There is both some research and real product development going on to incorporate this type of behavior and its management into languages and operating systems. The current technique to maintain independence for these tool solutions is to build the necessary reflective information (time and priority) into application message protocols. A more expensive (in maintenance and execution overhead) but more general solution is to use a customized dynamic message delivery mechanism with built in support for constraints.

Another aspect of real-time systems is the strong need for adaptable configuration management. (Configuration almost by definition is reflective.) Real-time systems change over time as plants are changed or redesigned for new production. To make matters worse these systems often are not allowed to stop or restart. Online configuration and even online upgrades or installations are required. In order to do these changes safely, these systems must understand themselves, i.e. know their configuration and module versions. The configuration and versions are the reflective attributes and the configuration and versioning methods the reflective behavior.

An important area for all systems, which is now beginning to be addressed, is that of practical techniques for the specification and construction of reusable designs, often referred to as "frameworks". Through the use of object frameworks, prepackaged and reusable applications can be built and applied to many different but similar situations or problems. Reflection is the key to this technology. In order to reuse a frameworks, information about the connections and relationships amongst the objects must be captured. If frameworks are to be adaptable to changing circumstances they must be aware of their own structure. A simple macro is a simple framework, but macros are usually implemented in a manner which deposits them into the software as individually connected objects, but without the knowledge of their macro-ness. Intelligent, reflective frameworks maintain their identity as a framework and can be managed at that level. Reconfiguration software can copy, move and change the framework because its structure is identifiable and described in the software (the framework object).

Computations can involve multiple objects where the desired behavior is the aggregate of the behavior of the participating objects. This sometimes is referred to as a contract amongst the objects to provide a service. There are two versions of this interaction: 1) a group of objects perform a series of methods to effect a requested action; this, perhaps more accurately, could be considered an object transaction, 2) a group of objects act as a whole, exhibiting a singular behavior. Reflective structures are required in these groupings in order to manage their interactions. Take the first case, a multi-object transaction. If something happens in the middle of the series of interactions, some corrective action must be taken to ensure system integrity. Half-completed transactions are not allowed. This means the transaction itself is an object and most of its behavior is reflective, that is, its job is to manage the transaction. The second case is more difficult and may also be related to frameworks in addition to contracts. The aggregate as a whole must have an identity. In some systems this can become a new class of object, but in a distributed system this is not easy to do (perhaps impossible in some situations); therefore, the aggregate itself is an encapsulating object which provides reflective attributes and behavior about the contracting objects.

This leads to the need in many systems especially real-time ones, to tolerate or adapt to system faults such as those encountered in software (a failed interaction) and hardware (a destroyed component). For example, a broken object transaction must provide mechanisms equivalent to the rollback functionality of data base systems. In order to do this, the object contracts must be manifest (objects) possibly preconfigured with recovery points. This is recovery management, a reflective behavior.

In the case of hardware, the system should in software (objects which model hardware) reflect the existence of the hardware and its configuration. With this knowledge automatic recoveries can be made from hardware failures. Service engineers or verification software can verify system configurations.

Tolerant software often means redundancy such as hot backups or replicated servers. To utilize these redundancies, the system must be aware of their existence and serviceability. This awareness is a reflective attribute and the logic to manage its use, reflective behavior.

One technique used in both fault tolerance and dynamic configuration software is that of migration. Objects may be moved from faulty or failing machines, or configuration software may redistribute objects and modules for performance or economy reasons. Again, system self-knowledge supports the movement of these objects in a manner which ensures that clients of the object's services are reconnected or error handled to avoid safety problems.

Through the understanding of reflective attributes and behaviors, and the use of practical approaches to architecture and implementation, a richly reflective system can be built to address complex applications such as real-time and multimedia systems.