

Using UML to Model Complex Real-Time Architectures*

Bran Selic
ObjecTime Limited
Kanata, Ontario, CANADA

1 Introduction

The term “software architecture” is widely used but infrequently specified. We define it as: *the organization of significant software components interacting through interfaces, those components being composed of successively smaller components and interfaces*¹.

Note that, when dealing with architectures, we are only interested in *high-level* features. Architecture necessarily involves abstraction — the elimination of irrelevant detail so that we can better cope with complexity.

The term “organization” pertains to the high-level structure of a system: its decomposition into parts and their mutual relationships (interaction channels, hierarchical containment, etc.). However, architecture is more than just structure. It includes rules on how system functionality is achieved across the structure. This includes high-level end-to-end behavioral sequences by which a system realizes its use cases.

Structure and behavior come together in interfaces, which, appropriately, also play a fundamental role in the specification of a software architecture. The last part of the definition above is simply a recursion of the first part. This tells us that architecture is a relative concept. It does not just occur at the “top” level of decomposition. A system may be decomposed into a set of subsystems, each of which, in turn, can be viewed as a system in its own right etc.

Software architectures play pivotal role in two types of situations. During initial development of a software system, an architecture is used to separate responsibilities and distribute work across multiple development teams. If the architecture is well defined, it should be straightforward to put the individually developed parts together and complete the system. Unfortunately, properly specified architectures are rare, which explains why there is a large incidence of so-called “integration” problems. (Almost invariably, an integration problem is the result either of an inadequate architecture or an inadequately specified architecture.)

*Reprint of abstract from *OMER Workshop Proceedings*, Peter Hofmann and Andy Schüerr (eds.), Bericht Nr. 1999-01, University of the Federal Armed Forces Munich, Neubiberg, Germany

¹I owe this definition to Grady Booch

However, even more importantly, software architectures play a crucial role in system evolution. It can be safely said that the architecture is the fundamental determinant of a system's capacity to undergo evolutionary change. A flexible architecture with loosely coupled components is much more likely to accommodate new feature requirements than one that has been highly optimized for just its initial set of requirements.

2 Requirements for Architectural Modeling

Complex systems often have complex architectures. An architectural modeling language must have the range to define all the necessary elements of that in a clear and unambiguous manner.

Most complex software systems are structured in some layered fashion. Layering is one of the most common ways of dealing with complexity. Yet, no standard programming language used in industry provides the concept of a "layer" as a first class concept. Instead, layering is simulated in a variety of ways, such as the use of compilation module boundaries, none of which are adequate to precisely capture the subtle semantics of layering. For instance, although layered architectures are often depicted as simple "onion-peel" structures, the layering in most real-time systems is far more complicated. Consider, for example, the well-known seven-layer architecture of ISO's Open System Interconnection standard. This architecture deals exclusively with communication aspects of a system. It is typically implemented as an application on top of an operating system, which represents an orthogonal layer hierarchy. In other words, complex systems require multiple dimensions of layering — two are never enough.

Another issue associated with architectures is the potential for reuse. Many modern systems are built around the "product line" concept. A product line is a set of different products that are built around a common abstract architecture. This architecture is then refined in different ways to realize individual products. This leads us to the requirement for subclassing at the architectural level. Needless to say, with very poor support for high-level concepts such as layering, no standard programming language in use today supports such a capability.

It is clear that we need a new breed of specification languages that is capable of addressing these requirements.

3 An Architectural Modeling Language

We define an architectural modeling language building on the ideas of the ROOM modeling language [SGW94]. A part of ROOM was specifically designed for modeling architectures of complex real-time systems. However, we will expand on those ideas and, furthermore, express them in the industry standard Unified Modeling Language (UML) [BRJ98b] [BRJ98a] [OMG97b] [OMG97a]. This allows us to take advantage of seman-

tics and notation that are widely recognized by software practitioners. Specifically, we use the extensibility mechanisms² of UML: stereotypes, tagged values, and constraints. In other words, we define our specific architectural modeling concepts as specializations of generic UML concepts. These specializations, usually expressed as stereotypes, conform to the generic semantics of the corresponding UML concepts, but provide additional semantics specified by constraints.

3.1 Capsules

The basic concept is that of an architectural object called a *capsule*. A capsule is a stereotype of the UML class concept with some specific features:

- it is always an active object, which means that it has behavior that executes concurrently with other behaviors
- it has an encapsulation shell such that it not only hides the implementation from outside observers, but also prevents the implementation from directly (and arbitrarily) accessing the external environment; in other words, it is an encapsulation shell that works in both directions³
- it may be a truly distributed object — it may even span multiple physical nodes; this makes it suitable for modeling physically distributed conceptual entities.

Capsules are used to capture major architectural components of real-time systems. For instance, a capsule may be used to capture an entire subsystem, or even a complete system. As we shall see later on, a capsule may encapsulate any number of internal capsules.

One of the interesting features of capsules is that they can have multiple interfaces, called *ports*. Each interface presents a distinct aspect of a capsule. Different collaborators can access different interfaces, possibly in parallel. We shall describe ports in more detail in the following section.

A capsule uses its port for *all* interactions with its environment. The communication is done using signals, which can be used to carry both synchronous and asynchronous interactions. The advantage of signals as the underlying communication vehicle is that, in contrast to communications based on procedure calls, they are more amenable to distribution.

Because capsules are a kind-of class, they can be subclassed. This gives us the capability to produce variations and refinements of architectural components and even entire architectures.

²The term “extensibility mechanism” is somewhat misleading since they are really used not to introduce new concepts, but to allow the definition of specialized versions of existing ones.

³In classical OO languages, this is not the case: while the implementation is hidden from external entities, the implementation has unhindered access to the environment. This creates problems since a “component” in some class library may be very tightly coupled with other entities. Unfortunately, this coupling can only be determined by careful inspection of the implementation.

3.2 Ports

In contrast to interfaces one finds in traditional object-oriented programming languages, ports are distinct objects (stereotypes of the UML class concept). They convey signals between the environment and the capsule. The types of signals and the order in which they may appear is defined by the *protocol* associated with the port. Protocols are discussed in the next section.

Ports are used not only for receiving incoming signals, but also for sending all outgoing signals. This means that a capsule is fully encapsulated: internal components never reference an external entity, but only communicate through ports. Consequently, a capsule class is fully self contained — which makes it a truly reusable component.

3.3 Protocols

A *protocol* specifies a set of valid behaviors (signal exchanges) between two or more collaborating capsules. However, to make such a dynamic pattern reusable, protocols are de-coupled from a particular context of collaborating capsules and are defined instead in terms of abstract entities called *protocol roles*. A protocol role represents the activities of one participant of a protocol. Formally, it is defined by a set of incoming signals, a set of outgoing signals, and an optional behavior specification. The behavior specification represents that subset of the behavior specified for the overall protocol that directly involves this role.

A particularly common type of protocols are *binary* protocols, which involve only two roles. For binary protocols it is sufficient to define one protocol role to define the entire protocol.

The relationship between ports and protocols is crucial: each port plays exactly one role in some protocol. The protocol role of a port represents the type of the port.

Protocols are modeled in UML as a stereotype of the collaboration concept. Since collaborations are generalizable elements, they too can be refined using inheritance-like mechanisms to produce variations and refinements.

Protocol roles are stereotypes of the classifier role concept in UML.

3.4 Connectors

To allow capsules to be combined into aggregates, we define the concept of a *connector*. A connector is an abstraction of a message-passing channel that connects two or more ports. Each connector is typed by a protocol that defines the possible interactions that can take place across that connector.

Connectors are stereotypes of the association class concept in UML with the added constraints that they can only pass signals and that their behavior is governed by a protocol.

3.5 Composite Capsules

Using the simple concepts of capsules, ports, and connectors, we can easily assemble complex aggregates of diverse capsules that achieve complex functionality. The design paradigm behind this is very similar to the design of hardware. Complex systems emerge by interconnecting simpler specialized parts drawn from a components catalog.

It is often the case that we may need to reuse a particular object composition pattern in a variety of different situations. In other words, we would like to make these assemblies into components in their own right. For this purpose, the object paradigm gives us the class concept. A particularly convenient way of realizing such a class is to define it as a capsule. This makes the capsule concept recursive: a capsule can be decomposed into lesser capsules, and so on. There are no theoretical limits to this hierarchy, capsules can be nested to whatever extent is necessary to realize our desired system. Note that such composite capsules can have their own ports, like any other capsules. These ports may be used to delegate functionality, using an internal connector, to some internal component. Such ports are called *relay* ports since their function is simply to act as a funnel for signal traffic. Alternatively, a port may be connected directly to a state machine. This type of port maintains a queue of incoming signals that have been received but not yet processed by the state machine. These ports are called *end* ports.

A composite capsule, therefore, represents a network of collaborating capsules. A particularly important feature of composite capsules is their *assertion semantics*. What this means is that when a composite capsule is instantiated, all of its internal nested parts are *automatically* instantiated along with it. Furthermore, since the internal capsules may themselves be composites, an entire system can be created with just one single “create” action. The consequence of assertion semantics is that the designer is liberated from the often tedious and highly error prone task of having to write the explicit code that generates the complex internal structure piece by piece. This not only saves effort, but also provides a tremendous boost in reliability, because in many dynamic systems, the code used to establish a structure can represent a major percentage of the overall software.

4 Summary

Architecture plays a key role in the design of complex real-time systems. In order to specify the types of architectures that are common in this domain, we need a suitable set of modeling concepts. We have presented here a very simple set of modeling concepts consisting of capsules, ports, protocols, and connectors, that has already been proven effective in industrial practice. To make these modeling capabilities available to the broadest set of users possible and to take advantage of the commercial tools, we have expressed them

using the industry standard Unified Modeling Language. For this purpose, we have used the extensibility mechanisms of UML. These allowed us to very effectively capture the full semantics of the concepts in a manner consistent with the general semantics of UML.

5 Acknowledgment

The author recognizes the major contribution of Jim Rumbough of Rational Software Corporation who collaborated very closely with the author on the work reported here.

References

- [BRJ98a] G. Booch, J. Rumbough, and I. Jacobson. *The Unified Modeling Language Reference Manual*. Addison Wesley Publishing Co., 1998.
- [BRJ98b] G. Booch, J. Rumbough, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Publishing Co., 1998.
- [OMG97a] OMG. *UML Notation Guide (Version 1.1)*. The Object Management Group, Framington MA, 1997. Doc. ad/97-08-05.
- [OMG97b] OMG. *UML Semantics (Version 1.1)*. The Object Management Group, Framington MA, 1997. Doc. ad/97-08-04.
- [SGW94] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, New York, 1994.
- [SR98] B. Selic and J. Rumbough. Using UML for Modeling Complex Real-Time Systems. ObjecTime Limited/Rational Software Corp. white paper, March 1998.