Using UML for Modeling Complex Real-Time Systems

Bran Selic

ObjecTime Limited, 340 March Rd.
Kanata, Ontario, Canada K2K 2E4
bran@objectime.com

Abstract. Real-time software systems encountered in telecommunications, aerospace, and defense often tend to be very large and extremely complex. It is crucial in such systems that the software has a well-defined architecture. This not only facilitates construction of the initial system, it also simplifies system evolution. We describe a set of modeling constructs that facilitate the specification of complex software architectures for real-time systems. These constructs are derived from field-proven concepts originally defined in the ROOM modeling language. Furthermore, we show how they can be represented using the industry-standard Unified Modeling Language (UML) by using the powerful extensibility mechanisms of UML.

1 Introduction

Perhaps the only characteristic common to all real-time software systems is *timeliness*; that is, the requirement to respond correctly to inputs within acceptable time intervals. Beyond that, the term "real-time" covers a surprisingly diverse spectrum of systems, ranging from purely time-driven to purely event-driven systems, and from soft real-time systems to hard real-time systems. Each of these different categories of systems has highly specialized idioms, proven design patterns, and modeling styles.

We focus on a major subset of real-time systems that are characterized as complex, event-driven, and, possibly distributed. Complexity implies not only magnitude but also significant diversity. Such systems are most frequently encountered in telecommunications, aerospace, defense, and automatic control applications. The effort required to design and realize these systems typically involves large development teams. Because of the high initial development cost, when major new requirements are identified for these systems, their software tends to be modified rather than rewritten.

Under such circumstances an overriding concern is the *architecture* of the software. This refers to the essential structural and behavioral framework on which all other aspects of the system depend. This is the software equivalent of the load-bearing frames in buildings – any changes to this foundation necessitates complex and costly changes to substantial parts of the system. A well-designed architecture not only

simplifies construction of the initial system, but more importantly, it promotes evolution.

To facilitate the design of good architectures, it is extremely useful to capture the proven architectural design patterns of the domain as first-class modeling constructs. As our primary source for this we have selected the Real-time Object-Oriented Modeling Language (ROOM) [1]. ROOM is an architectural definition language developed specifically for complex real-time software systems. Its architectural modeling constructs have been in use for over a decade and have proven their effectiveness in hundreds of different large-scale industrial projects.

We have chosen to represent these constructs using the industry-standard Unified Modeling Language (UML) [2] [3] to obtain the benefits of a popular and widely-supported notation. UML has proven very well suited to this purpose since it has built-in extensibility mechanisms that allow domain-specific extensions [4]. With these mechanisms it was possible to represent the necessary constructs relying exclusively on standard UML semantics.

In effect, the modeling constructs described in this document represent a type of library of applied UML concepts intended primarily for use in modeling the architectures of complex real-time systems. They can be used in combination with the more basic UML modeling concepts and diagrams to provide a powerful and comprehensive modeling facility.

2 Modeling Structure

The *structure* of a system identifies the major components of that system and the relationships between them (communication relationships, containment relationships, etc.). For our purposes, we define three principal constructs for modeling structure:

- capsules
- ports
- connectors

Capsules correspond to the ROOM concept of *actors*. They are complex, physical, possibly distributed architectural objects that interact with their surroundings through one or more signal-based boundary objects called ports. A *port* is a physical part of the implementation of a capsule that mediates the interaction of the capsule with the outside world—it is an object that implements a specific interface. Each port of a capsule plays a particular role in a collaboration that the capsule has with other objects. To capture the complex semantics of these interactions, each port has an associated *protocol* that defines the valid flow of information (signals) through that port. In a sense, a protocol captures the contractual obligations that exist between capsules. Protocols are discussed in more detail later in the document. By forcing capsules to communicate solely through ports, it is possible to fully de-couple their internal implementations from any direct knowledge about the environment. This makes them highly reusable.

Connectors, which correspond to ROOM *bindings*, are abstract views of signal-based communication channels that interconnect two or more ports. The ports bound by a connection must play mutually complementary but compatible roles in a protocol. If we abstract away the ports from this picture, connectors really capture the key communication relationships between capsules. These relationships have architectural significance since they identify which capsules can affect each other through direct communication.

The functionality of simple capsules is realized directly by the state machine associated with the capsule. More complex capsules combine the state machine with an *internal* network of collaborating *sub-capsules* joined by connectors. These sub-capsules are capsules in their own right, and can themselves be decomposed into sub-capsules. This type of decomposition can be carried to whatever depth is necessary, allowing modeling of arbitrarily complex structure. The state machine (which is optional for composite capsules), the sub-capsules, and their connections network represent parts of the *implementation* of the capsule and are hidden from external observers.

2.1 Ports

Ports are objects whose purpose is to act as boundary objects for a capsule instance. They are "owned" by the capsule instance in the sense that they are created along with their capsule and destroyed when the capsule is destroyed. Each port has its identity and state that are distinct from the identity and state of their owning capsule instance (to the same extent that any part is distinct from its container).

Each port plays a specific role in some protocol. This *protocol role* defines the *type* of the port, which simply means that the port implements the behavior specified by that protocol role.

Because ports are on the boundary of a capsule, they may be visible both from outside the capsule and inside. When viewed from the outside, all ports present the same impenetrable object interface and cannot be differentiated except by their identity and the role that they play in their protocol. However, viewed from within the capsule, we find that ports can be either *relay ports* or *end ports*. These two differ in their internal connections—relay ports are connected to sub-capsules while end ports are connected directly to the capsule's state machine. Generally speaking, relay ports serve to selectively export the "interfaces" of internal sub-capsules while end ports are boundary objects for the state machine of a capsule. Both relay and end ports may appear on the boundary of the capsule and, as noted, are indistinguishable from the outside.

Relay ports. *Relay ports* are ports that simply pass all signals through. They provide an "opening" in the encapsulation shell of a capsule that can be used by its sub-capsules to communicate with the outside world without actually being exposed to the outside world (and vice versa). A relay port is connected, through a connector, to a sub-capsule and is normally also connected from outside to some other "peer"

capsule. They receive signals coming from either side and simply relay it to the other side keeping the direction of signal flow.

Relay ports allow the direct (zero overhead) delegation of signals destined for a capsule to a sub-capsule without requiring intervention by the state machine of the capsule. Relay ports can only appear on the boundary of a capsule and, consequently, always have *public* visibility.

End Ports. To be useful, a chain of connectors attached via relay ports must ultimately terminate in an end port that is attached to a state machine. End ports are boundary objects for the state machines of capsules. They are the ultimate sources and sinks of all signals sent between capsules. To send a signal, a state machine invokes a send or call operation on one of its end ports. The signal is then relayed through the attached connector, possibly passing through one or more relay ports and connectors, until it finally strikes another end port in a different capsule. The end port has a queue to hold asynchronous messages that have been received but not yet processed by the state.

Like relay ports, end ports may appear on the boundary of a capsule with public visibility. These ports are called *public end ports*. Such ports are boundary objects of both the state machine and the containing capsule. However, end ports may also appear completely inside the capsule as part of its internal implementation structure. The state machine uses these ports to communicate with its sub-capsules or with external implementation-support layers. Internal end ports are called *protected end ports* since they have protected visibility.

End ports that are connected to supporting layers are called *service access points*. Implementation support *layers* represent run-time services, such as operating system services, that may be shared by multiple capsules and, hence, cannot be incorporated directly into any particular one of the capsules. This is used to model *run-time layering* relationships.

UML Modeling. In UML terms, a port object is modeled by the «*port*» stereotype, which is a stereotype of the UML Class concept. As noted earlier, the type of a port is defined by the protocol role played by that port. Since protocol roles are abstract classes, the actual class corresponding to this instance is one that *implements* the protocol role associated with the port. In UML the relationship between the port and the protocol role is referred to as a "realizes" *relationship*. The notation for this is a dashed line with a solid triangular arrowhead on the specification end (Figure 1). It is a form of generalization whereby the source element—the port—inherits only the behavior specification of the target—the protocol role—but not its structure.

A capsule is in a composition relationship with its ports. (This means that ports cannot exist independently of their capsule.) If the multiplicity of the target end of this relationship is greater than one, it means that multiple instances of the port exist at run time, each participating in a separate instance of the protocol. If the multiplicity is a range of values, it means that the number of ports can vary at run time and that ports can be dynamically created and destroyed (possibly subject to constraints).

Notation. In UML class diagrams, the ports of a capsule are listed in a special labeled list compartment as illustrated in Figure 2. The *ports* list compartment normally appears after the attribute and operator list compartments. This notation takes advantage of the UML feature that allows the addition of specific named compartments.

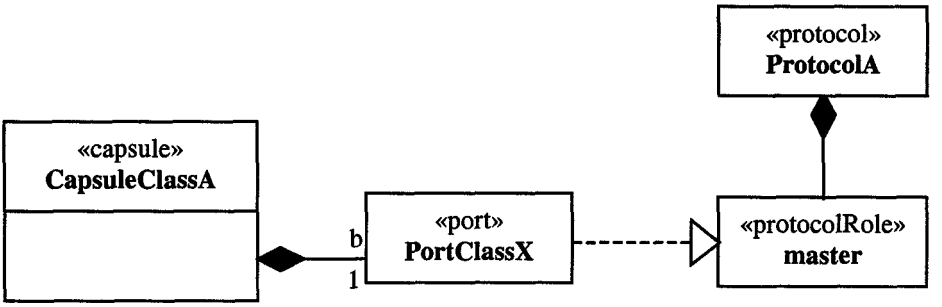

Fig. 1. The relationship between capsules, ports, protocols, and protocol roles in UML notation

All external ports (relay ports and public end ports) have public visibility while internal ports have protected visibility (e.g., port b2 in Figure 2). The protocol role (type) of a port is normally identified by a pathname since protocol role names are unique only within the scope of a given protocol. For the most frequent case of protocols involving just two parties (binary protocols), a simpler notational convention is used: a suffix tilde symbol ("~") is used to identify the conjugated protocol role (e.g., port b2) while the base role name is implicit with no special annotation (e.g., port b1). Ports with a multiplicity other than 1 have the multiplicity factor included between square brackets. For example, port b1[3] has a multiplicity factor of exactly 3 whereas a port designated by b5[0..2] has a variable number of instances not exceeding 2.

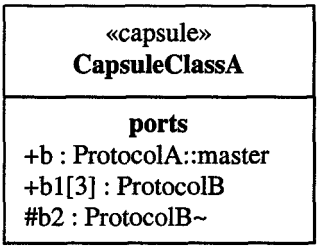

Fig. 2. Special list compartment listing all the ports attached to a capsule class in a UML class diagram

Figure 2 shows how ports are indicated in class diagrams. However, they also appear in collaboration diagrams. In these diagrams, objects are represented by the appropriate classifier roles—sub-capsules by *capsule roles* and ports by *port roles*. To

reduce visual clutter, port roles are generally shown in iconified form, represented by small black or white squares (Figure 3). Public ports are represented by port role icons that straddle the boundary of the corresponding capsule. This shorthand notation allows them to be connected both from inside and outside the capsule without unnecessary crossing of lines and also identifies them clearly as boundary objects.

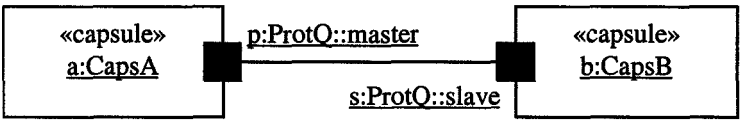

Fig. 3. Iconified notation for ports in collaboration diagrams

For the case of binary protocols, the port playing the conjugate role is indicated by a white-filled (versus black-filled) square. In that case, the protocol name and the tilde suffix are sufficient to identify the protocol role as the conjugate role; the protocol role name is redundant and can be omitted. Similarly, the use of the protocol name alone on a black square indicates the base role of the protocol. This convention makes it easy to spot when complementary protocol roles are connected.

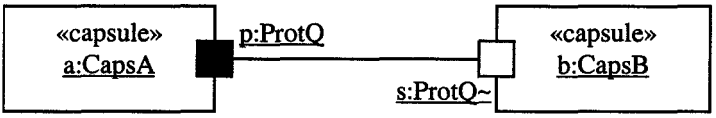

Fig. 4. Notation for the port roles of binary protocols

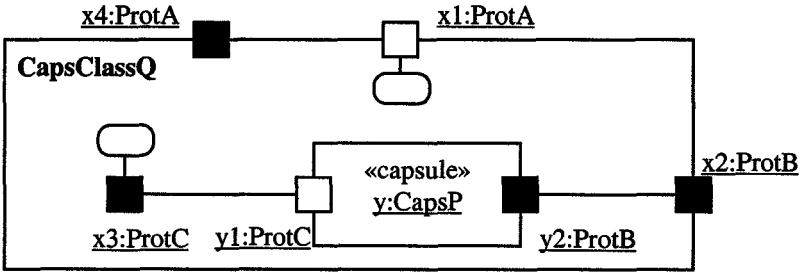

Fig. 5. Port notation in a collaboration diagram (internal capsule view)

When the decomposition of a capsule is shown, we can see the implementation inside the capsule using a UML collaboration diagram (Figure 5). In this case, an end port is distinguished by a small rectangle with rounded corners ("roundtangle") that symbolizes the state machine behind the end port. For example, port x1 is a public (conjugated) end port and port x3 is a protected end port. Ports without a roundtangle that are bound to a sub-capsule's port (e.g., port x2) or another end port are relay

ports. Ports that are not connected to either a state machine roundtangle or to a sub-capsule port, such as port x4 above, are *indeterminate.*

2.2 Connectors

A connector represents a communication channel that provides the transmission facilities for supporting a particular signal-based protocol. A key feature of connectors is that they can only interconnect ports that play complementary roles in the protocol associated with the connector.

The similarity between connectors and protocols might suggest that the two concepts are equivalent. However, this is not the case, since protocols are abstract specifications of desired behavior while connectors are physical objects whose function is merely to convey signals from one port to the other. Typically, the connectors themselves are passive conduits.

UML Modeling. In UML, a connector is modeled directly by a standard UML association. This association is defined between two or more ports of the corresponding capsule classes. In a collaboration diagram, it is represented as an association role. No UML extensions are required for representing connectors.

2.3 Capsules

Capsules are the central architectural modeling construct. They represent the major architectural elements of complex real-time systems. Typically, a capsule has one or more ports through which it communicates with other capsules. It cannot have operations or public parts other than ports, which are its exclusive means of interaction with the external world. As noted, a capsule may contain one or more sub-capsules joined together by connectors. This internal structure is specified as a collaboration.

A capsule may have a state machine that can send and receive signals via the end ports of the capsule and that has control over certain elements of the internal structure such as the optional dynamic creation and destruction of sub-capsules.

The Internal Structure. A capsule's *complete* internal decomposition, that is, its implementation, is represented by a collaboration. This collaboration includes a specification of all of its ports, sub-capsules, and connectors. Like ports, the sub-capsules and connectors are strongly owned by the capsule and *cannot* exist independently of the capsule. They are automatically created when the capsule is created and automatically destroyed when their capsule is destroyed.

Some sub-capsules in the structure may not created at the same time as their containing capsule. These may be created subsequently, when and if necessary, by the state machine of the capsule.

An important feature for modeling complex dynamically created structures is the concept of so-called *plug-in* roles in a collaboration diagram. These are placeholders

for sub-capsules that are filled in *dynamically* at run time. A collaboration diagram with plug-ins represents a dynamic architectural template or pattern. Plug-ins are necessary because it is not always known in advance which specific objects will play those roles. Once this information is available, the appropriate capsule instance (which is owned by some other composite capsule) can be "plugged" into such a slot and the connectors joining its ports to other sub-capsules in the collaboration automatically established. When the dynamic relationship is no longer required, the capsule is "removed" from the plug-in slot, and the connectors to it are taken down.

The Capsule State Machine. The optional state machine associated with a capsule is just another part of a capsule's implementation. However, it has certain special properties that distinguish it from the other constituents of a capsule:

- It cannot be decomposed further into sub-capsules.
- There can be at most one such state machine per capsule, although each sub-capsule can have its own state machine. Capsules that do not have state machines are simple containers for sub-capsules.
- It responds to signals arriving on an end port of a capsule and can send signals through those ports.
- It acts as a controller of all sub-capsules. It can create and destroy sub-capsules that are identified as dynamic, and it can plug in and remove external sub-capsules into the plug-in slots.

UML Modeling. In UML, a capsule is represented by the «*capsule*» stereotype of Class. The capsule is in a composition relationships with its ports, sub-capsules (except for plug-ins), and internal connectors. This means that they only exist while their capsule is instantiated. Except for public ports, all the various capsule parts, including sub-capsules and connectors, have protected visibility.

The internal structure of a capsule is modeled by a UML collaboration. Sub-capsules are indicated by appropriate sub-capsule (classifier) roles. Plug-in slots are also identified by sub-capsule roles. The type of the sub-capsule for a plug-in slot identifies the set of protocol roles (pure interface type) that must be satisfied by capsules that can be plugged into the slot.

Dynamically created sub-capsules are indicated simply by a variable multiplicity factor. Like plug-in slots, these may also be specified by a pure interface type. This means that, at instantiation time, any implementation class that supports that interface can be instantiated. This provides for genericity in structural specifications.

Notation. The class diagram notation for capsules uses standard UML notational conventions. Since it is a stereotype, the stereotype name may appear above the class name in the name compartment of the class rectangle. An optional special icon associated with the stereotype may appear in the upper right-hand corner of the name compartment (Figure 6). Sub-capsules are indicated by composition associations while plug-ins are rendered through aggregation relationships. Alternatively, the structure of a class can be shown as a collaboration as in Figure 7.

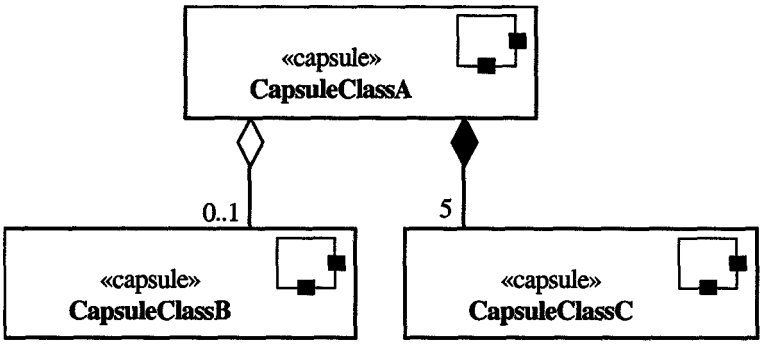

Fig. 6. Capsule notation - class diagram view

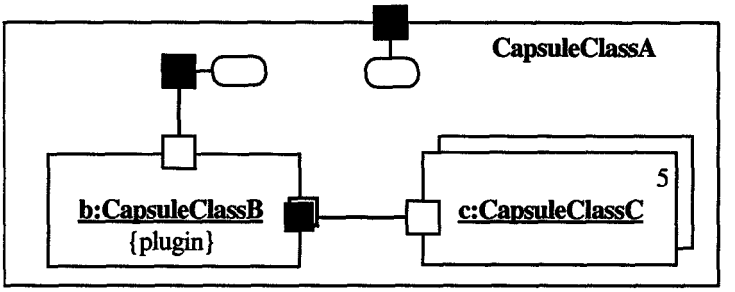

Fig. 7. Collaboration diagram view of the capsule class shown in Figure 6

3 Modeling Behavior

Behavior at the architectural level is captured using the concept of protocols.

3.1 Protocols

A protocol is a specification of desired behavior that can take place over a connector—an explicit specification of the contractual agreement between the participants in the protocol. It is pure behavior and does not specify any structural elements. A protocol consists of a set of participants, each of which plays a specific role in the protocol. Each protocol role is specified by a unique name and a set of signals that are received by that role as well as the set of signals that are sent by that role (either set could be empty). As an option, a protocol can also have a specification of the valid communication sequences; a state machine can be used to specify this. Finally, a protocol may also have a set of prototypical interaction sequences (these can

be shown as sequence diagrams). These must conform to the protocol state machine, if one is defined.

Binary protocols, involving just two participants, are by far the most common and the simplest to specify. One advantage of these protocols is that only one role, called the *base role*, needs to be specified. The other, called the *conjugate*, can be derived from the base role simply by inverting the incoming and outgoing signal sets. This inversion operation is known as *conjugation.*

UML Modeling. A protocol role is modeled in UML by the *«protocolRole»* stereotype of ClassifierRole. A protocol is modeled by the *«protocol»* stereotype of Collaboration with a composition relationship to each of its protocol roles. This collaboration does not have any internal structural aspects (i.e., it has no association roles).

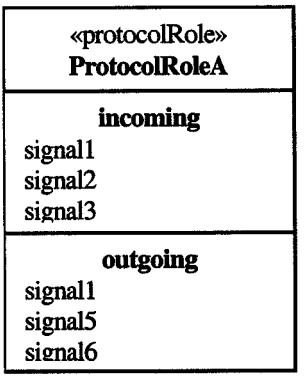

Fig. 8. Protocol role notation in class diagrams

Notation. Protocol roles can be shown using the standard notation for classifiers with an explicit stereotype label and two optional specialized list compartments for incoming and outgoing signal sets, as shown in Figure 8. The state machine and interaction diagrams of a protocol role are represented using the standard UML state machines.

4 Summary

We have defined a set of structural modeling concepts that are suitable for modeling the architectures of complex real-time systems. They were derived from similar constructs defined originally in the field-proven ROOM modeling language. The semantics and notation for representing these constructs are based on the industry-standard Unified Modeling Language. The result is a domain-specific "library" of pre-defined UML patterns that can be used directly as first-class modeling constructs.

An important aspect of these modeling constructs is that they have fully formal semantics. The definition of the static and dynamic semantics of the individual modeling constructs is outside the scope of this document but can be found in reference [1]. This means that models created using these constructs can be formally verified for correctness. Furthermore, they can be used to construct executable models to achieve early validation of high-level design and even analysis models. Finally, such models can be, *and have been*, used to automatically generate complete implementations. This bypasses the error-prone and lengthy step of manually converting a design model into a programming language realization.

Acknowledgements

The author would like to express his gratitude to James Rumbaugh of Rational Software whose contribution to the work described in this paper was instrumental. In addition, Grady Booch and Ivar Jacobson (also of Rational) provided invaluable and crucial feedback and suggestions on optimal ways to render these constructs in UML.

References

1. Selic, B., Gullekson, G., and Ward, P.: Real-Time Object-Oriented Modeling. John Wiley & Sons, New York, NY (1994)
2. OMG: UML Semantics. Version 1.1. The Object Management Group, Doc. no. ad/97-08-04. Framingham MA. (1997)
3. OMG: UML Notation Guide. Version 1.1. The Object Management Group, Doc. no. ad/97-08-05. Framingham MA. (1997)
4. OMG: UML Extension for Objectory Process for Software Engineering. Version 1.1. The Object Management Group, Doc. no. ad/97-08-06. Framingham MA. (1997)