Using UML for Modeling Complex Real-Time Systems

Bran Selic, ObjecTime Limited
Jim Rumbaugh, Rational Software Corporation
March 11, 1998

Abstract

The embedded real-time software systems encountered in applications such as telecommunications, aerospace, and defense typically tend to be large and extremely complex. It is crucial in such systems that the software is designed with a sound architecture. A good architecture not only simplifies construction of the initial system, but even more importantly, readily accommodates changes forced by a steady stream of new requirements. In this paper, we describe a set of constructs that facilitate the design of software architectures in this domain. The constructs, derived from field-proven concepts originally defined in the ROOM modeling language, are specified using the Unified Modeling Language (UML) standard. In particular, we show how these architectural constructs can be derived from more general UML modeling concepts by using the powerful extensibility mechanisms of UML.

1. Introduction

We apply the Unified Modeling Language (UML [2], [3]) to describe a set of constructs suitable for modeling an important category of real-time software systems. They are derived from the set of concepts initially defined in the ROOM modeling language [1].

1.1 The Application Domain

The one common feature of all real-time software systems is *timeliness*; that is, the requirement to respond correctly to inputs within acceptable time intervals. However, this ostensibly simple property characterizes a vast spectrum of very different types of systems ranging from purely time-driven to purely event-driven systems, from soft real-time systems to hard real-time systems, and so on. Over time, each of these categories of systems has developed its own idioms, design patterns, and modeling styles that collectively capture the distilled experience of many projects.

In this document we focus on a major category of real-time systems that are characterized as complex, event-driven, and, potentially, distributed. Such systems are most frequently encountered in telecommunications, aerospace, defense, and automatic control applications. The size and complexity of these systems demand a considerable initial development effort, typically involving large development teams, that is followed by an extended period of evolutionary growth. During this time new requirements are identified and the system is modified incrementally to meet them.

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. Therefore, a well-designed architecture is not only one that simplifies construction of the initial system, but more importantly, one that easily accommodates changes forced by new system requirements.

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. This is why ROOM, among other things, incorporates concepts that constitute a domain-specific *architectural definition language*. These constructs have been in use for over a decade and have proven their effectiveness across hundreds of diverse large-scale industrial projects. We have found that this domain-specific usage can be cleanly implemented using the general-purpose UML.

1.2 The Use of UML

As noted earlier, UML was used to capture these constructs. We found it very well suited to this purpose so that *no new UML modeling concepts were needed*. The standard UML tailoring mechanisms based on stereotypes, tagged values, and constraints were used to great effect. For example, the central "actor" concept of ROOM is captured by the «capsule» stereotype – a specialization of the general UML concept of a class. This stereotype adds additional domain-specific semantics onto the various aspects that are associated with a UML class (e.g., state machines,

collaborations). The stereotype facility provides a simple shorthand way of referring to the collection of well-formedness rules that specify these 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 are used in combination with all the other UML modeling concepts and diagrams to provide a comprehensive modeling toolset.

1.3 Acknowledgements

We would like to express our gratitude to Grady Booch and Ivar Jacobson who reviewed drafts of this document and provided valuable feedback and many helpful suggestions on optimal ways to render these constructs in UML.

2. Modeling Constructs and Notation

In this section we examine the most important modeling constructs used for representing complex real-time systems and also describe how they are captured and rendered using UML. The constructs can be partitioned into two major groups: constructs for modeling structure and constructs for modeling behavior.

2.1 Modeling Structure

The *structure* of a system identifies the entities that are to be modeled and the relationships between them (e.g., communication relationships, containment relationships). UML provides two fundamental *complementary* diagram types for capturing the logical structure of systems: class diagrams and collaboration diagrams. Class diagrams capture universal relationships among classes—those relationships that exist among instances of the classes in all contexts. Collaboration diagrams capture relationships that exist only within a particular context—a pattern of usage for a particular purpose that is not inherent in the class itself. Collaboration diagrams therefore include a distinction between the usage of different instances of the same class, a distinction captured in the concept of *role*. In the modeling approach described here, there is a strong emphasis on using UML collaboration diagrams to explicitly represent the interconnections between architectural entities. Typically, the complete specification of the structure of a complex real-time system is obtained through a *combination* of class and collaboration diagrams.

Specifically, 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*[1] boundary objects[2] 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, ports are associated with a *protocol* that defines the valid flow of information (signals) between connected ports of capsules. In a sense, a protocol captures the contractual obligations that exist between capsules. Protocols are discussed in more detail in the section dealing with behavior (2.2.1). 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.

---

[1] In distributed systems where one cannot generally assume shared address spaces, signal-based interactions are more appropriate than operation-based interactions. Note that, in UML, signal-based interactions can model both synchronous and asynchronous communications.
[2] The term "boundary" object is used in the same sense as in the *UML Extension for Objectory Process for Software Engineering[4]*; that is, an object that "lies on the periphery of a system, but within it". Ports represent a particular type of boundary object.

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. In collaboration diagrams, they are represented by association roles that interconnect the appropriate ports. 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. Ports are included to allow the encapsulation of capsules under the principles of information hiding and separation of concerns.

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 structures with just this basic set of structural modeling constructs. 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.

We now describe each of these constructs in more detail.

2.1.1 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).

Although ports are boundary objects that act as interfaces, they do not map directly to UML interfaces. A UML interface is purely a behavioral thing—it has no implementation structure. A port, on the other hand, includes both structure and behavior. It is a composite part of the structure of the capsule, not simply a constraint on its behavior. It realizes the architectural pattern that we might call "manifest interface".

As noted earlier, 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, when viewed from within the capsule, we find that ports can be one of two kinds: *relay ports* and *end ports*. They differ in their internal connections—relay ports are connected to sub-capsules while end ports are connected 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.

2.1.1.1 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. This is achieved without delay or loss of information unless there is no connector attached on the other side. In the latter case, the signal is lost.

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.

2.1.1.2 End Ports

To be useful, a chain of connectors must ultimately terminate in an end port that communicates with a state machine. End ports are boundary objects for the state machines of capsules (although, as we shall see, some of them also serve as boundary objects for capsules as well). End ports are the ultimate sources and sinks of all signals sent by capsules. These signals are generated by the state machines of 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 chained connectors, until it finally encounters another end port, usually in a different capsule. Since signal-based communication can be asynchronous, an end port has a queue to hold messages that have been received but not yet processed by the state machine (i.e., it acts as a mailbox). The reception of the signal and the dispatching of the receiving state machine is performed by the state machine according to standard UML semantics [2].

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[3]. However, end ports may also appear completely inside the capsule as part of its internal implementation structure. These ports are used by the state machine to communicate with its sub-capsules or with external *implementation-support layers*[4]. These internal end ports are called *protected end ports* since they have protected visibility.

Note that the kind of port is totally determined by its internal connectivity and its visibility outside the capsule; the various terms (relay port, public end port, private end port) are merely shorthand terminology. A public port that is not connected internally may become either a relay port or an end port depending on how it is later connected, or it may remain unconnected and be a sink for incoming signals.

---

[3] There is nothing conceptually deep behind this; it is simply a shortcut to avoid defining an extra relay port.

[4] 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 layering relationships and may be considered part of the implementation infrastructure.

2.1.1.3 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. 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. 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).

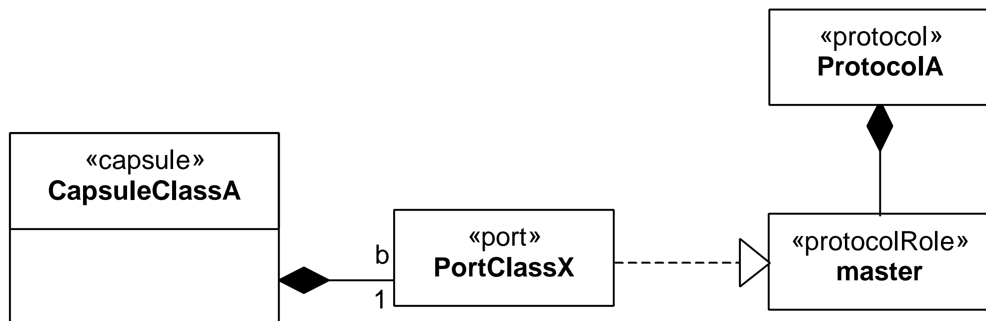

Figure 1. Ports, protocols, and protocol roles

Figure 1 shows an example of a single port named b belonging to capsule class CapsuleClassA. This port realizes the master role of the protocol defined by protocol class ProtocolA. Note that the actual port class, PortClassX, being an implementation class that may vary from implementation to implementation, is normally not of interest to the modeler until the implementation stage. Instead, the information that *is* of interest is the protocol role that this port implements. For this reason and also for reasons of notational convenience, the notation shown in Figure 1 is not normally used and is replaced by the more compact form described in the following section.

2.1.1.4 Notation

In 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.

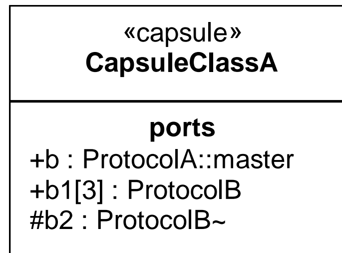

Figure 2. Port notation - class diagram representation

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 example, in Figure 2, port b plays the master role defined in the protocol class called ProtocolA. For the very frequent case of binary protocols, a simpler notational convention is used: a suffix tilde symbol ("~") is used to identify the conjugated[5] 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.

Figure 2 shows how ports are shown in class diagrams. However, they are also depicted in the collaboration diagrams that describe the internal decomposition of a capsule. 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. Public ports are represented by port role icons that straddle the boundary of the corresponding capsule roles as shown in Figure 3. 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.

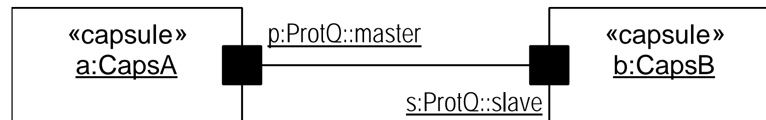

Figure 3. Port notation - collaboration diagram

Note that the labels in Figure 3 are adornments to the port roles and should not be confused with association end names of the connector[6]. Since ports are uniquely identified by their names, as a graphical convenience it is possible to arrange the public port roles around the perimeter of a sub-capsule box in any order. This can be used to minimize crossovers between connector lines.

For the case of binary protocols, an additional stereotype icon can be used: 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 should be omitted. Similarly, the use of the protocol name alone on a black square indicates the

[5] For a definition of these terms, please refer to section 2.2.1.
[6] One distinction is that port role names are underlined while association end names are not.

base role of the protocol. For example, if the “master” role in protocol ProtQ (Figure 3) is declared as the base, then the diagrams in Figure 4 and Figure 3 are equivalent. This convention makes it easy to see when complementary protocol roles are connected.

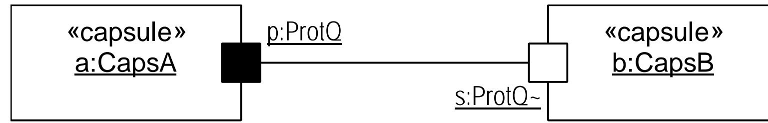

Figure 4. Notational conventions for binary protocols

Ports with a multiplicity factor that is greater than one can also be indicated graphically using the standard UML multiobject notation as shown in Figure 5. This is not mandatory (the multiplicity string is sufficient) but it emphasizes the possibility of multiple instances of the port.

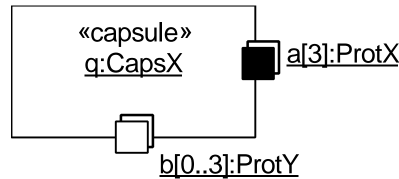

Figure 5. Ports with multiplicity factor greater than 1

The examples of graphical notation for ports in collaboration diagrams discussed to this point are all based on external (opaque) views of capsules. In those cases, it is not possible or even desirable to determine whether a port is a relay port or an end port. However, when the decomposition of a capsule is shown, we can see inside the capsule and the end port/relay port distinction is indicated graphically as shown in the example in Figure 6.

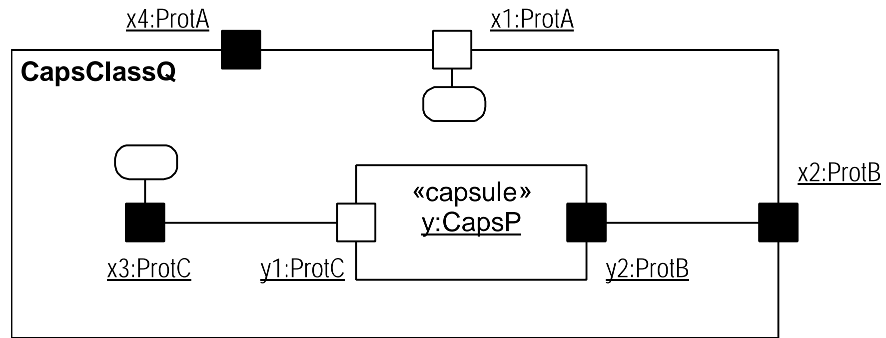

Figure 6. Port notation - collaboration diagram (internal view)

In this case, end port icons are shown as being connected to a small “roundtangle” that iconically symbolizes the state machine of the capsule. For example, port x1 is a public (conjugated) end port and port x3 is a protected end port. (Note that even though there are multiple state machine icons shown in a collaboration diagram, there is still at most one state machine per capsule. The end port may all be connected to a single state machine icon, but often the connections would be graphically messy so the icon can be repeated.) Relay ports are identified by

the fact that they are connected to a port on a sub-capsule (e.g., port x2). Ports that are not connected to either a state machine roundtangle or to a sub-capsule port, such as port x4 above, are *indeterminate*. This means that their classification into either end or relay ports has been deferred and is typically resolved in a sub-class, or they may simply remain unconnected.

2.1.2 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. In principle, the protocol roles do not necessarily have to belong to the same protocol, but in that case they have to be compatible with the protocol of the connector[7].

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. (In practice, physical connectors may sometimes deviate from the specified behavior. For example, as a result of an internal fault, a connector may lose, reorder, or duplicate messages. This type of failure is common in distributed communication channels.)

2.1.2.1 UML Modeling

In UML, a connector is modeled by an association and is declared through an association role in a capsule collaboration diagram. This association exists between two or more ports of the corresponding capsule classes. (For advanced applications in which the connector has physical properties, an association *class* is used since the connector is actually an object with a state and an identity. As with ports, the actual class that is used to realize a connector is an implementation issue.) The relationship to the supported protocol is implicit through the connected ports. Consequently, no UML extensions are required for representing connectors.

2.1.2.2 Notation

Since connectors do not require any special stereotypes, they do not require any notational extensions. In a class diagram, an association may be used to represent a connector. Whether or not a connector is rendered in a class diagram, it must be explicitly shown in the collaboration diagram of the capsule class to which it belongs. A binary protocol connector is shown as a line drawn between two complementary port roles. Examples of binary protocol connectors can be seen in Figure 3 and Figure 6 above.

Connectors for tertiary and higher-order protocols are represented using the standard UML diamond notation for n-ary associations. However, high-order protocols have not yet been implemented in ROOM so experience with them is still needed.

[7] We will not discuss the rules of protocol compatibility here except to say that it is based on behavioral sufficiency.

2.1.3 Capsules

Capsules are the central modeling construct in this approach. They are used to 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 (UML) 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. Hence, this state machine may be regarded as implementing reflective behavior, that is, behavior that controls the operation of the capsule itself.

In addition to ports, capsules have some specific static and dynamic constraints on semantics that specializes them from general classes. We describe the most important ones in the following sections.

2.1.3.1 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 created when the capsule is created and destroyed when their capsule is destroyed.

Some sub-capsules in the structure may not created at the same time as their containing capsule. Instead, they may be created subsequently, when and if necessary, by the state machine of the capsule. The state machine can also destroy such capsules at any time. This follows the UML rules on composition.

The structure of a capsule may contain so-called *plug-in* roles. These are, in effect, placeholders for sub-capsules that are filled in *dynamically*. This is necessary because it is not always known in advance which specific objects will play those roles at run time. Once this information is available, the appropriate capsule instance (which is owned by some other composite capsule) can be "plugged" into such a slot[8] and the connectors joining its ports to other sub-capsules in the collaboration are 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.

Dynamically created sub-capsules and plug-ins allow the modeling of dynamically changing structures while ensuring that all valid communication and containment relationships between capsules are specified explicitly. This is key in ensuring architectural integrity in a complex real-time system.

2.1.3.2 The 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:

[8] In the fully dynamic case, this is achieved by actions of the capsule's state machine. However, it is also possible for this to be done prior to run time (e.g., at compilation time or system load time) if that information is available.

- It cannot be decomposed further into sub-capsules. It specifies behavior directly. State machines, however, can be decomposed into hierarchies of simpler state machines using standard UML capabilities.
- There can be at most one such state machine per capsule (although sub-capsules can have their own state machines). Capsules that do not have state machines are simple containers for sub-capsules.
- It handles signals arriving on any end port of a capsule and can send signals through those ports.
- It is the only entity that can access the internal protected parts in its capsule. This means that it acts as the *controller* of all the other sub-capsules[9]. As such, it can create and destroy those sub-capsules that are identified as dynamic, and it can plug in and remove external sub-capsules as appropriate.

2.1.3.3 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-in sub-capsules), 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. To distinguish plug-in "sub-capsules", a special tagged value is defined for each sub-capsule role:

- *isPlugIn* – a Boolean value that, if true, indicates that this is a plug-in slot, otherwise it is a regular sub-capsule.

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.

Despite its additional restrictions, the state machine associated with a capsule is modeled by the standard link between a UML Classifier and a State Machine. The implementation/decomposition of a capsule is modeled by a standard UML collaboration element that can be associated with a classifier.

2.1.3.4 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

[9] Note that this is the only part that has explicit knowledge of other parts or other sub- capsules. Hence, it is not reusable outside the context defined by the capsule decomposition and is an integral part of the capsule.

associated with the stereotype may appear in the upper right-hand corner of the name compartment. Both of these can be seen in the class diagram shown in Figure 7 (although showing both in the same class box is unnecessary). Also, if a capsule has ports, the notation may include a port list compartment also shown in Figure 2.

Sub-capsules in a class diagram may be indicated by composition relationships as shown in the class diagram in Figure 7, with plug-in sub-capsules indicated by aggregation relationships.

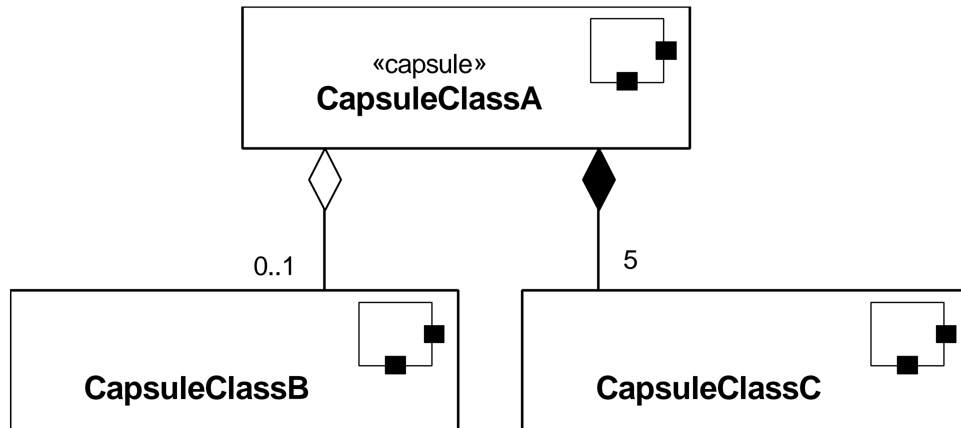

Figure 7. Capsule notation – class diagram

The decomposition of a capsule can also be depicted using the UML notation for composite objects as shown in Figure 6 and Figure 8 below. In addition to the information shown in the class diagram, this notation also specifies the precise interconnection topology between sub-capsules, indicated by connectors. The state machine part is not shown explicitly in these diagrams (nor in any UML class diagram) but is implied by the small white roundtangles next to the end ports. Plug-in sub-capsules are indicated the keyword "{plugin}" (e.g., sub-capsule b in Figure 8).

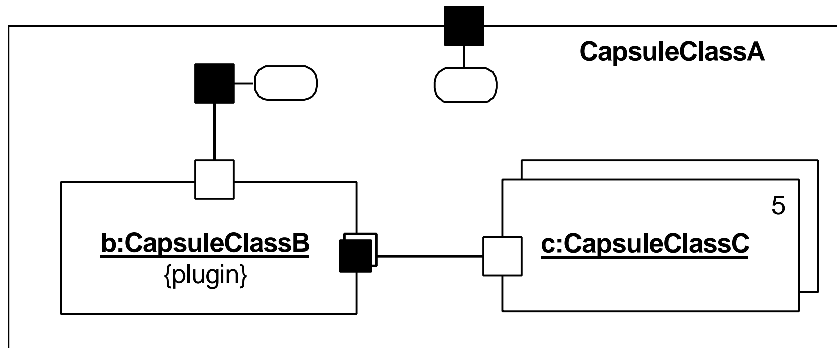

Figure 8. Collaboration diagram corresponding to Figure 7

2.2 Modeling Behavior

Behavior at the architectural level is captured using the concept of protocols. In this section we also examine two other behavior-related ROOM features. While standard UML state machines are used for modeling the state machines of capsules, we examine a particular form of usage that, in combination with inheritance, strongly

facilitates re-use. Finally, we look at a specific way of modeling time and timing services that reflect the way such things are handled in the vast majority of real-time systems.

2.2.1 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 comprises a set of participants, each of which plays a specific role in the protocol. Each such *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 may 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*.

2.2.1.1 UML Modeling

A protocol role is modeled in UML by the «*protocolRole*» stereotype of ClassifierRole. This stereotype has two dependencies to Signal: one for incoming and one for outgoing signals. (This is a property of UML classes in general). Like any classifier, it may also have an associated state machine that captures the local behavior of the protocol role. This state machine has to be compliant with the protocol state machine.

A protocol is modeled in UML by the «*protocol*» stereotype of Collaboration with a composition relationship to each of its protocol roles representing the standard relationship that a collaboration has with its “owned elements”. This collaboration does not have any internal structural aspects (i.e., it has no association roles). Like all generalizable elements[10], a protocol can be refined using standard inheritance. The state machine and collaborations associated with a protocol are inherited directly from Classifier.

2.2.1.2 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 9. The state machine and interaction diagrams of a protocol role are represented using the standard UML notation.

[10] Although currently Collaborations are not subtypes of Classifiers, this is a defect in the definition of UML 1.1. This is expected to be rectified in the next release.

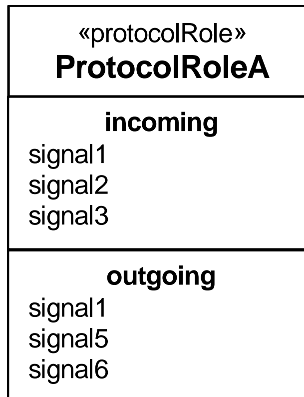

Figure 9. Protocol role notation - class diagram

A special shorthand notation is provided for binary protocols since they are by far the most common. As noted earlier, for binary protocols, only the base role needs to be specified. Furthermore, since the role state machine and the protocol state machine are the same in this case, only the protocol state machine needs to be defined. For this reason, the notation for binary protocols combines elements of the protocol role notation by including directly the incoming and outgoing signal lists with the protocol class. The protocol stereotype and its corresponding icon (Figure 10) help to differentiate this from the protocol role notation.

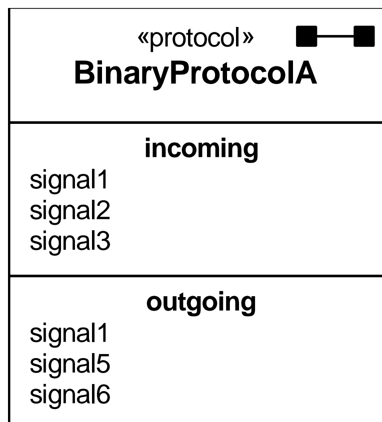

Figure 10. Notation for binary protocols – class diagram

Finally, a protocol usage may also be indicated with a standard collaboration use diagram represented by a dashed oval with dashed lines for each of its roles.

2.2.2 State Machines

The specification for the state machine part of a capsule as well as the specification of valid protocol sequences is done using standard UML state machines. In this section we demonstrate some specific ways in which state machines can be used with the structural modeling constructs already described.

2.2.2.1 Port-Based Triggers

In practice, it often happens that two or more ports of the same capsule use the same protocol but are semantically distinct. Also, the same signal may appear in more than one protocol role supported by different ports of a capsule. In either case, it may be necessary to distinguish the specific end port that received the current signal. That allows applications to handle the same signal differently depending on the source of that signal as well as the state. We refer to this type of trigger as a *port-based trigger*.

UML Modeling

We assume that an implementation can provide the information about which port received the signal as one of the parameters of an event. In that case, port-based triggers can then be modeled in UML easily by using guard conditions that checks for a particular source port. Hence, no special conceptual extensions are necessary to support this capability.

2.2.2.2 The Abstract Behavior Pattern

In this section we describe a common and very useful heuristic that takes advantage of the hierarchical nature of UML state machines. It was first used with ROOMcharts[11], the ROOM-based variant of state charts, and is captured here as a usage pattern called the *abstract behavior pattern*.

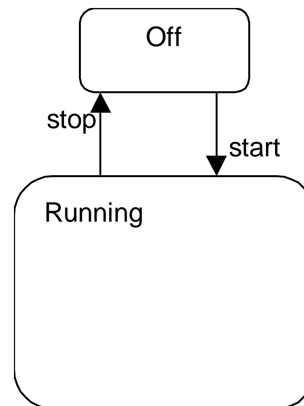

Figure 11. An abstract state machine

Consider the abstract state machine in Figure 11. The term “abstract” here denotes that this is a state machine that needs to have more detail added before it can be used for practical purposes. For instance, the simple state machine in Figure 11 is representative of the most abstract level of behavior (the “control” automaton) of many different types of elements in real-time systems. Although they all share this high-level form, the different element types may have widely different detailed behaviors in the Running state depending on their purpose. Therefore, this state machine would most likely be defined in some abstract capsule class that serves as the root class for the different specialized capsule classes

[11] ROOMcharts were used to describe the state machines associated with ROOM actors, the ROOM equivalent of the capsule concept described here.

Let us therefore define two such different refinements of this abstract state machine, using inheritance. These two refinements, R1 and R2, are shown in Figure 12. For clarity, we have drawn the elements inherited from the parent class using a gray pen.

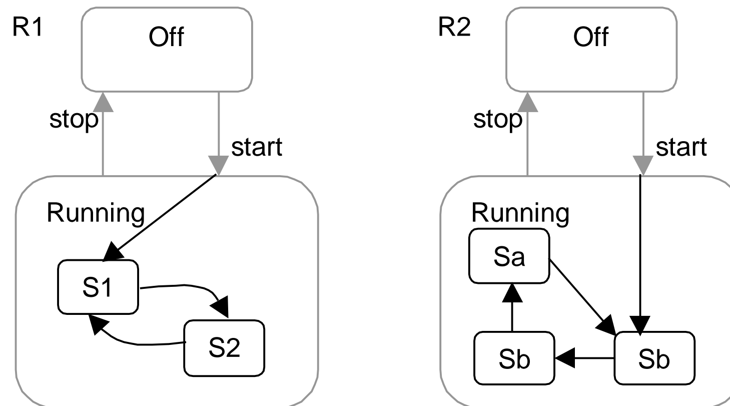

Figure 12. Two refinements of the state machine in Figure 11

The two refinements clearly differ in how they decompose the Running state and also how they extend the original “start” transition. These choices can only be made, of course, once the refinement is known and, hence, could not have been done with a single end-to-end transition in the abstract class.

UML Modeling

The ability to “continue” both incoming transitions and outgoing transitions is fundamental for the type of refinement described above. It may seem that entry points and final states, combined with continuation transitions are sufficient to provide these semantics. Unfortunately, this is not sufficient when there are multiple different transitions that need to be extended.

What is required for the abstract behavior pattern is a way of chaining two or more transition segments that are all executed in the scope of a single run-to-completion step. This means that transitions entering a hierarchical state are split into the incoming part that effectively terminates on the state boundary and an extension that continues within the state. Similarly, outgoing transitions emanating from a hierarchically nested state are segmented into a part that terminates on the enclosing state boundary and a part that continues from the state boundary to the target state. This effect can be achieved in UML with the introduction the *chain state* concept. This is modeled by a stereotype *(«chainState»)* of the UML State concept. This is a state whose only purpose is to “chain” further automatic (triggerless) transitions onto an input transition. A chain state has no internal structure—no entry action, no internal activity, no exit action. It also has no transitions triggered by events. It may have any number of input transitions. It may have an outgoing transition with no trigger event; this transition automatically fires when an input transition activates the state. The purpose of the state is to chain an input transition to a separate output transition. Between the input transition(s) and the chained output transition, one connects to another state inside the containing state and the other connects to another state outside the containing state. The purpose of introducing a chain state is to separate

the internal specification of the containing state from its external environment; it is a matter of encapsulation.

In effect, a chain state represents a “pass through” state that serves to chain a transition to a specific continuation transition. If no continuation transition is defined, then the transition terminates in the chain state, and some transition on an enclosing state must eventually fire to move things along.

The example state machine segment in Figure 13 illustrates chain states and their notation. Chain states are represented in a state machine diagram by small white circles located within the appropriate hierarchical state (this notation is similar to initial and final states, which they resemble). The circles are stereotype icons of the chain state stereotype and are usually drawn near to the boundary for convenience. (In fact, a notational variation would be to draw these on the border of the enclosing state similar to the notation for ports.)

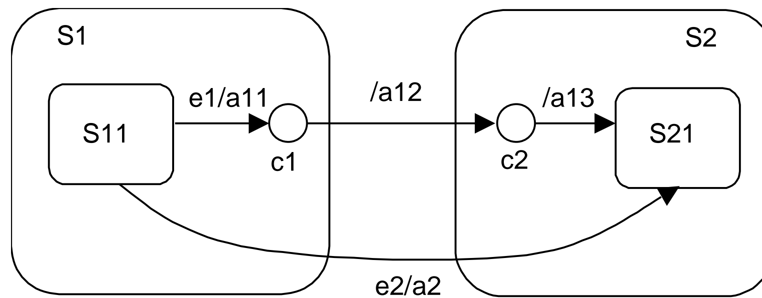

Figure 13. Chain states and chained transitions

The chained transition in this example consists of the three chained transition segments e1/a11–/a12–/a13. When signal e1 is received, the transition labeled e1/a11 is taken, its action a11 executed, and then chain state c1 is reached. After that, the continuation transition between c1 and c2 is taken, and finally, since c2 is also a chain state, the transition from c2 to S21. If the states along these paths all have exit and entry actions, the actual sequence of action executions proceeds as follows:

- exit action of S11
- action a11
- exit action of S1
- action a12
- entry action of S2
- action a13
- entry action of S21

All of this is executed in the scope of a single run-to-completion step.

This should be compared against the action execution semantics of the direct transition e2/a2, which are:

- exit action of S11
- exit action of S1
- action a2
- entry action for state S2
- entry action for state S21

2.2.3 Time Service

As can be expected, in most real-time systems time is a first-order concern. In general, two forms of time-based situations need to be modeled: the ability to trigger activities at a particular *time of day* and, the ability to trigger activities after a certain *interval* has expired from a given point in time.

Most real-time systems require an explicit and directly accessible (controllable) timing facility—a *time service*. This service, which can be accessed through a standard port (service access point), converts time into events that can then be handled in the same way as other signal-based events. For example, with such a service, a state machine can request that it be notified with a “timeout” event when a particular time of day has been reached or when a particular interval has expired.

2.2.3.1 UML Modeling

The concept of a timing service does not require any UML extensions or special notation. Service access points for accessing the timing service are simply indicated by protected end ports whose type is defined as “TimeServiceSAP” (a system-defined protocol role).

3. Well-Formedness Rules and Dynamic Semantics

It is extremely useful to define both the static semantics (i.e., the rules for well-formedness) and the dynamic (run-time) semantics of the real-time constructs described here. With these semantic rules fully defined and consistent, and in conjunction with an action specification language that is also complete and consistent (e.g., C++ or Ada) that is used to specify the details of state-transition actions and object methods, the resulting models will be *executable*. Furthermore, if all the necessary detail is included in a model, such a model can be used to automatically generate complete implementations. This approach has proven quite successful with ROOM and the ObjecTime Developer toolset since it can significantly reduce development time and product reliability compared to more traditional development models.

The definition of the static and dynamic semantics of the individual modeling constructs is outside the scope of this document.

4. Summary of UML Extensions

4.1 Stereotypes

The following stereotypes are defined as part of this extension

Metamodel Class	Stereotype Name
Collaboration	protocol
ClassifierRole	protocolRole
Class	port
Class	capsule
State	chainState

4.2 Tagged Values

This extension does not introduce any new tagged values other than those that are an integral part of the domain-specific stereotypes listed above.

4.3 Constraints

This extension does not introduce any new constraints other than those associated with the well-formedness rules of the domain-specific stereotypes listed above.

4.4 Prerequisite Extensions

This extension requires no other extensions to UML for its definition.

References

[1] Selic, B., Gullekson, G., and Ward, P., "Real-Time Object-Oriented Modeling," John Wiley & Sons, New York, NY, 1994.

[2] "UML Semantics," version 1.1 (1 September 1997), The Object Management Group, doc. no. ad/97-08-04.

[3] "UML Notation Guide," version 1.1 (1 September 1997), The Object Management Group, doc. no. ad/97-08-05.

[4] "UML Extension for Objectory Process for Software Engineering" version 1.1 (1 September 1997). The Object Management Group, doc. no. ad/97-08-06.