UML for Real-Time Overview

Andrew Lyons

April 1998

Abstract

This paper explains how the ***Unified Modeling Language (UML)***, and powerful modeling constructs originally developed for the modeling of complex real-time systems in the ***Real-Time Object-Oriented Modeling language (ROOM)***, have been combined into ***UML for Real-Time***. It is directed at developers of complex real-time software systems (e.g., telecommunications, aerospace, defense, and automatic control applications).

Introduction

There are unique challenges faced in real-time software development. Every real-time software developer recognizes that the requirement for latency, throughput, reliability, and availability are far more stringent than for general purpose, or business software. For real-time system developers, understanding the impact of design decisions and effectively communicating functionality can be a daunting task. 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.

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. **UML for Real-Time** combines UML, role modeling, and ROOM concepts to deliver a complete solution for modeling complex real-time systems. UML, role modeling and ROOM are briefly described below.

UML is a general-purpose modeling language for specifying, visualizing, constructing and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. UML has a strong set of general purpose modeling language concepts applicable across domains.

Role modeling captures the structural communication patterns between software components. In UML 1.1, collaboration diagrams, which form the basis of structural design patterns, became first class modeling entities. ***ObjecTime Limited (ObjecTime)*** was a member of the UML 1.1 definition team and contributed the role modeling capabilities of ROOM to the UML standard.

ROOM is a visual modeling language with formal semantics, developed by ObjecTime. It is optimized for specifying, visualizing, documenting, and automating the construction of complex, event-driven, and potentially distributed real-time systems. It incorporates the role modeling concepts discussed in this document that enable the capture of architectural design patterns.

UML for Real-Time is a complete real-time modeling standard, co-developed by ObjecTime and Rational Corporation, that combines UML 1.1 modeling concepts, and special modeling constructs and formalisms originally implemented in ***ObjecTime Developer*** and defined in the ROOM language. ObjecTime Developer is a software automation tool that provides model execution capabilities, and automatically generates complete code for complex real-time applications from these modeling constructs. UML for Real-Time supports all of the automation capabilities that are available in ObjecTime Developer today.

Modeling Perspectives

As software systems become increasingly more complex, software architecture, and techniques for capturing it, become increasingly more important. In addition the architecture of these systems

can be viewed from many perspectives. UML for Real-Time represents a collection of best engineering practices that have proven successful in the modeling of large and complex real-time systems. Typically, the complete specification of the structure, and behavior, of a complex real-time system is obtained through a combination of model perspectives. This section explores role models and their relationship to class and instance models.

A ***class model*** is a high level generalization of a system. By defining the universal relationships of a set of classes to each other, it allows us to specify the possible systems that can be constructed from those classes. For example, an *OC48LineCard* class may always have a *slot* attribute or it may always be associated with an instance of an *OC48LineCardController* class. Class diagrams are used to capture class models. Class modeling focuses on relationships and class decomposition. Class models provide the 10,000-foot view of a design. It is not possible to determine specific communication relationships between instances from a class model.

An **instance model** is a low-level specialized view of a system. It captures the properties of *instances* of classes in an application. For example, an instance in a switch application may be an *OC48LineCard*. An *OC48LineCardController* may send it a reset message. Instance modeling often focuses on the interactions of instances (objects) required to satisfy a specific execution scenario. Interaction diagrams (i.e., collaboration or sequence diagrams) are used to capture instance models. Instance models provide the ground-level scenario based view of a design. The communication relationships between instances are derivable from the collection of interaction diagrams but are not present in a single instance diagram.

A **role model** is an intermediate-level view of a system. It is both a specialization of a class model, and a generalization of an instance model. It captures the properties of an application that are true for all instances of that application. It captures structural patterns that show in specific contexts, the roles classes play, and how they interact. For example in Figure 1, a *LineCardController* will always command a *LineCard.* The *LineCard* role may be realized by an instance of the class *OC48LineCard* or *OC192LineCard.*The *LineCardController* role may be realized by an *OC48LineCardController*, or *OC192LineCardController*. A role model is the 1,000-foot view of a design. It is more specific than a class model, but more general than an instance model. Collaboration diagrams are used to capture role models.

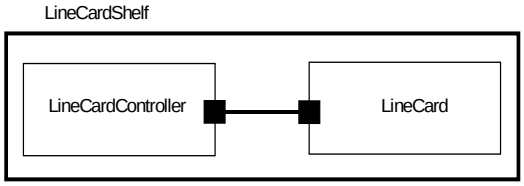

Figure 1 Class LineCardShelf collaboration diagram

Class, role and instance models are complementary; none are “better than” the others. As we start to analyze a system, instance models give us concrete examples. When we understand these we can abstract them to the role models which will represent all executions of our system. When we analyze a system, role models define the architecture. We can validate architecture using instance models. Role models can be used as specifications for more general class models or to directly generate systems. Class model libraries provide ready-made classes to implement role models.

ObjecTime Developer is the only real-time software development tool that implements the UML 1.1 collaboration role modeling constructs and the additional UML for Real-Time modeling constructs described in this document. Developers build graphical models to specify a system's components and their relationships (structure) and the system's response to events (behavior)

which can then be compiled directly into high performance C++ code in a process analogous to the way today's compilers produce machine-language code from higher-level languages.

UML for Real-Time

The modeling approach, described in this document, places a strong emphasis on using UML collaboration diagrams to explicitly represent structural design patterns. When the semantics of a UML metaclass must be refined, a new UML stereotype is introduced. This section introduces the important Capsule, Port, and Connector, stereotypes that have been introduced into UML for Real-Time to support the modeling of complex real-time systems. Figure 2 shows a UML for Real-Time collaboration diagram containing these three components.

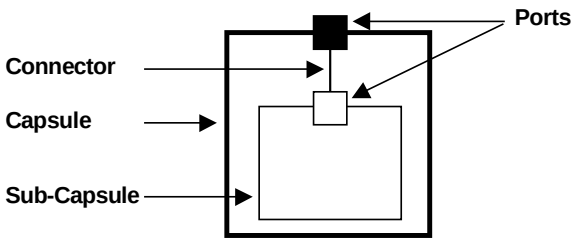

Figure 2 - A capsule collaboration diagram

Capsules correspond to the ROOM concept of *actors*. Capsules are complex, potentially concurrent, and possibly distributed active architectural components. They interact with their surroundings through one or more signal-based boundary objects called ports. Collaboration diagrams are used to describe the structural decomposition of a Capsule class.

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. Ports realize ***protocols***, which define the valid flow of information (signals) between connected ports of capsules. In a sense, a protocol captures the contractual obligations that exist between capsules. Because a protocol defines an abstract interface that is realized by a port, a protocol is highly reusable.

Ports provide a mechanism for a capsule to export multiple different interfaces; each tailored to a specific role. They also provide a mechanism to explicitly *connect* an exported interface of one capsule directly to the interface of another capsule. By forcing capsules to communicate solely through ports, it is possible to fully de-couple their internal implementations from any direct knowledge they have about the environment. This de-coupling makes capsules highly reusable.

Connectors 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 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.

Role modeling and class modeling

The combination of class and role modeling provides a very powerful modeling paradigm. Class modeling is focused on universal class relationships and class decomposition. Class diagrams are used to capture class models. Consider the class diagram of Figure 3. *CapsuleClassA* has a *CapsuleClassB* and a *CapsuleClassC* by value. The class diagram shows us the relationships between these classes, and could be annotated to show much richer content. Even though the example appears trivial, even with more associations and annotations it would be difficult to capture usage patterns in this view.

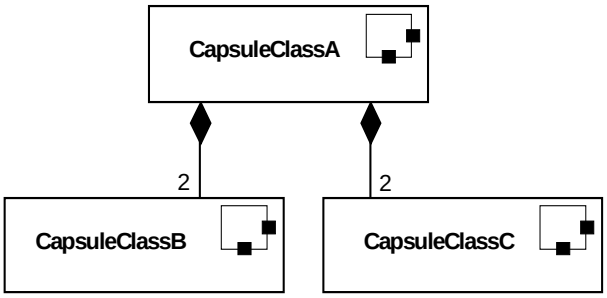

Figure 3 - Capsule class diagram

Collaboration diagrams can be used to capture structural usage patterns. Figure 4 shows us three of many possible collaboration diagrams for *CapsuleClassA*. A class can have only one structural-decomposition, so a single collaboration diagram is required to form a complete class specification (i.e., one must be chosen). The structure of *CapsuleClassA* can't be determined from its class diagram, however its class diagram can be derived from its structure.

Note: **b1**, **b2**, and **c1**, **c2** are roles in **CapsuleClassA** that can be played by **CapsuleClassB** and **CapsuleClassC** respectively. Optionally we could write **b1:CapsuleClassB**

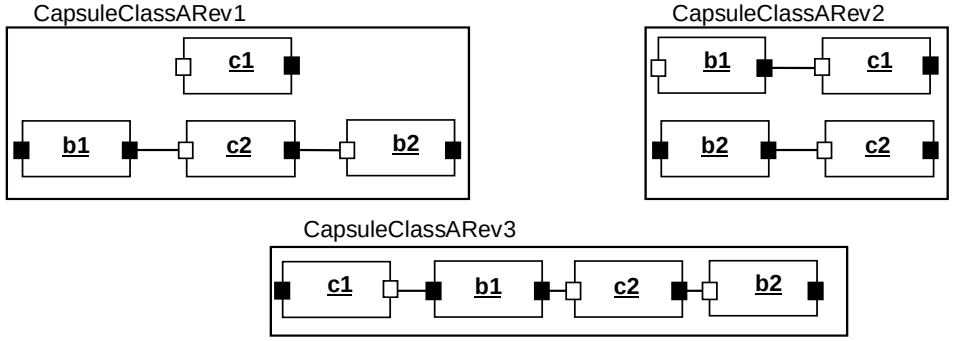

Figure 4- Several different possible collaborations for CapsuleClassA's implementation

Role modeling is focused on the roles class instances play in specific contexts. Collaboration diagrams and class diagrams are views of the same system from different perspectives, and at different levels of abstraction. Collaboration diagrams provide a mechanism to constrain a class diagram to precisely specified configurations. They are more specialized than class diagrams and more general than instance diagrams.

Capsule collaboration diagrams specify the roles Capsules play in different system contexts. For example, in Figure 4 CapsuleClassB plays a different role in each place it is used. Capsules communicate solely through ports and have no knowledge of the other Capsules in a collaboration in which they interact.

Mapping ROOM to UML for Real-Time

This section provides a summary of how ROOM structure charts, ROOMCharts, and ROOM terminology map into UML for Real-Time. The ROOM notation has been aligned with UML 1.1 but the semantics of ROOM models remain the same.

The advanced modeling capabilities of ROOM have been incorporated into UML for Real-Time and are available in the ObjecTime Developer toolset. A clear forward migration path, including loss-less model migration and continued technology enhancement, has been established to future versions of ObjecTime Developer.

Terminology

UML provides a very rich modeling environment and with it an established notation and nomenclature. As a result of the incorporation of ROOM concepts into UML for Real-Time, the ROOM terminology has been adjusted to avoid confusion.

The terms that have not changed their name or meaning are: Port, Relay Port, End Port, Protocol, Signal, State, Transition, Transition Segment, Entry Action, Exit Action, Initial Point, Guard, Trigger. Table 1 provides a summary of changed ROOM terminology and the corresponding equivalent in UML for Real-Time. Any terms not specifically mentioned have not changed.

ROOM term	**Equivalent UML for Real-Time term**
Actor	Capsule
Actor Reference	SubCapsule
Optional Actor	Optional[1] SubCapsule
Imported Actor	Plug-in SubCapsule
Actor Structure	Capsule Structure
Actor Behavior	Capsule Behavior
Replication Factor	Multiplicity
Port	Port Role[2]
Port Type	Protocol Role
Unconjugated protocol	Protocol Base Role
Conjugated protocol	Protocol Conjugated Role
Binding	Connector
Choice Point	Branch Point
Join-Point	Chain-State
Initial-Point	Initial-State
Data Class	Class
External Class	Class
MSC	Sequence Diagram

Table 1 - ROOM terminology changed in UML for Real-Time

Structure Diagrams

In UML for Real-Time, a capsule collaboration diagram is equivalent to a ROOM actor structure diagram. Figure 5 shows a simple ROOM actor structure diagram. Figure 6 shows the equivalent Capsule collaboration diagram in UML for Real-Time.

[1] Capsule optionality is specified using multiplicity. For example, in UML for Real-Time, a ROOM fixed actor reference (i.e.,Capsule) would be specified as *n*, while an optional or imported actor reference would be specified using *0..n*.

[2] ROOM protocols map into UML Protocols and ProtocolRoles. In ROOM we say that a protocol can be unconjugated or conjugated. In UML for Real-Time we say that a protocol role can represent the base, or conjugated role of a protocol, and that a port realizes a protocol role.

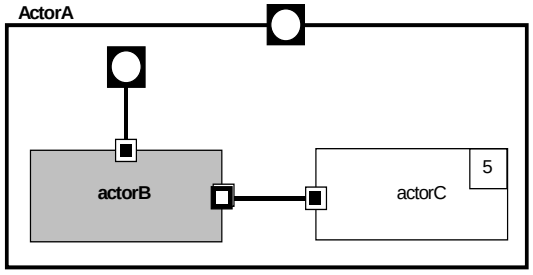

Figure 5 - A ROOM actor structure diagram

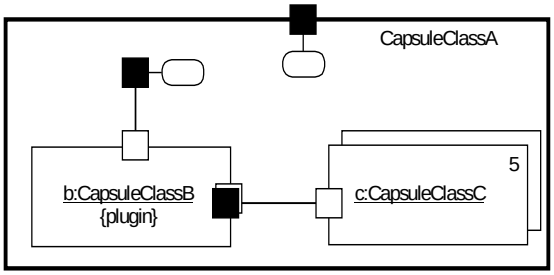

Figure 6 - A UML for Real-Time capsule collaboration diagram

Behavior Diagrams

In ROOM the implementation of an actor can be represented using a ROOMChart. In UML for Real-Time the implementation of a capsule class can be represented using an equivalent UML hierarchical finite state machine (FSM). Figure 7 shows a simple ROOMChart. Figure 8 shows the equivalent FSM in UML for Real-Time.

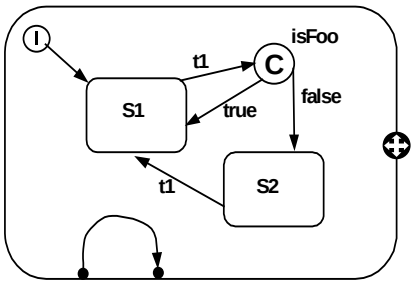

Figure 7- Example ROOMChart

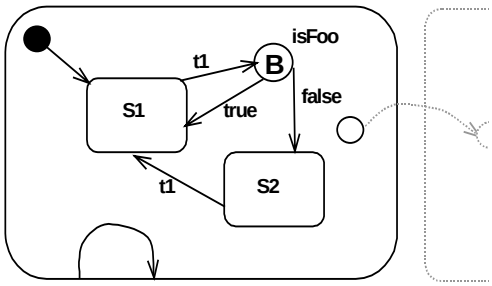

Figure 8- Example UML for Real-Time FSM

Summary

UML for Real-Time is an extension to the UML 1.1 visual modeling language that has been specifically fine-tuned for the development of complex, event-driven, real-time systems, such as those found in telecommunications, aerospace, defense, and automatic control applications.

Collaboration diagrams, which capture architectural design patterns, use the primary modeling constructs of capsules, ports and connectors to specify the structure of software components.

Capsules use ports to export their interfaces to other capsules. The functionality of simple capsules is realized directly by finite state machines, whose transitions are triggered by the arrival of messages on the capsule's ports. Capsules themselves can be decomposed into internal networks of communicating sub-capsules. The state machine and network of hierarchically decomposed sub-capsules allow the structural and behavioral modeling of arbitrarily complex systems.

Protocols are abstract interface specifications that are realized by ports. Capsules communicate with other capsules solely through ports; the internal implementation of a capsule is fully decoupled from any direct knowledge of its environment. This de-coupling makes capsules (and protocols) highly reusable.

UML for Real-Time is a visual modeling language that lets developers mirror the way real-time, event-driven systems actually work. These modeling constructs (i.e., role models, capsules, ports, connectors, and state machines) have rigorous formal semantics that provide for model execution, and can be used to generate code for complete, mission critical, real-time applications.

ObjecTime Developer is an object-oriented software-development tool that implements these UML for Real-Time concepts. All the benefits of UML 1.1 role modeling and the UML for Real-Time extensions, including design time model-visualization; run-time model animation of state machines and message flow (including message sequence charts); and run-time visual model, and source code level, debugging facilities; are available in ObjecTime Developer today. With ObjecTime Developer, all of the code for complete, high performance, mission critical applications is generated. The model is more than a model -- the model is the application.

About ObjecTime

ObjecTime Limited is the leading provider of visual development tools for the complex real-time systems market and the exclusive supplier of modeling and automatic code generation technology for the real-time domain to Rational Software Corporation. ObjecTime contributed to the UML 1.1 definition in the areas of formal specification, extensibility, and behavior, and played a key role in the definitions for state machines, common behavior, role modeling, and refinement.

ObjecTime Developer (a software automation tool which implements UML for Real-Time constructs) enables software developers to build applications using component-based visual design models. ***TotalCode™*** application generation automatically generates complete production quality C and C++ executables for UNIX, NT and a variety of real-time operating systems directly from system or component models. Application generation of fully or partially complete designs, plus animated visual and symbolic debuggers, encourage early and continuous design refinement and validation.

ObjecTime Developer is used for developing a wide variety of complex, real-time, event-driven applications in telecommunications, data communications, defense, aerospace, and other industries. The world's leading telecommunications, data communications, defense, aerospace and industrial control companies including Nortel, Lucent, Motorola, Lockheed-Martin, and Kodak. use ObjecTime Developer to accelerate real-time software delivery, and to improve the quality and functionality of their real-time products.

About the author

Andrew Lyons is a Senior Applications Specialist with ObjecTime Limited. He has been applying structured and object-oriented techniques and tools to the design of complex real-time software systems for seventeen years. He can be reached at andy@objectime.com.