Architectural Design of a Broadcasting System using UML-RT

Ingolf Krüger, Wolfgang Prenninger, Robert Sandner *

Technical University of Munich, D-80290 Munich, Germany
email: {kruegeri | prenning | sandnerr}@in.tum.de

Abstract

UML-RT provides graphical description techniques for modeling important aspects of software architectures for reactive and embedded real-time systems. One of its shortcomings is its restriction to binary communication links between components. Here, we show how to integrate the notion of broadcasting into architectural design with UML-RT. We introduce both a variant of sequence diagrams for graphically modeling broadcasting interaction patterns, and methodological guidelines for the systematic transformation of scenarios captured using these diagrams into a structural model for the system under development.

1 Introduction

The definition of an adequate software architecture is one of the decisive steps in the development process for complex distributed and reactive systems. In our view a system's software architecture comprises three central ingredients: the (hierarchical) decomposition of the system into components, the precise specification of the relationships (also called *interfaces*) between these components, and the forces and constraints that govern the chosen decomposition and definition of component relationships (cf., for instance, [BMR+96, DW98, Kru99] for other definitions of this term).

Given the importance of defining an adequate software architecture two key challenges arise: 1.) how to transfer the requirements, constraints, and forces captured for the system under consideration into a matching set of subcomponents with corresponding interfaces and connections, and 2.) how to document the selected architecture in a precise, yet transparent way such that the idea behind the architecture can be easily communicated to the developers of the system?

While these two challenges exist for architectural descriptions of arbitrary systems, embedded real-time systems typically pose additional problems; examples are the manifestation of requirements at timing behavior, of resource limitations, as well as of the underlying infrastructure for component communication within the software architecture.

UML-RT[SR98, Lyo98], a sequel to ROOM[SGW94], has been suggested as a notation for representing hierarchical structural decomposition, asynchronous binary component interactions via clear interfaces, and individual component behavior. The corresponding graphical description techniques available in UML-RT are capsule (and class) diagrams, sequence diagrams, and a subset of the UML's statecharts. These are significant aids in capturing important architectural aspects, and thus help addressing the second of the key challenges mentioned above.

*Our research was supported by the DFG within the priority program "SoftSpez" (SPP 1064) under project name *InTime*.

However, the binary communication model underlying UML-RT has its disadvantages in modeling real-world examples in the technical and embedded systems domain. Consider, for instance, the multicast and broadcast communication frequently used in automotive systems, avionics, and in mobile communications. This raises the question whether UML-RT is also an adequate means for architecture specifications despite its lack of explicit support for broadcasting.

In the remainder of this text we show how to use UML-RT effectively in developing software architectures for broadcasting systems; in particular, we will introduce systematic steps for performing the transition from captured requirements to a corresponding initial software architecture. Along the way we will cover two major topics. First, in Section 4, we introduce a new notation (similar to Message Sequence Charts and the UML's sequence diagrams) for capturing component interaction especially in broadcasting systems. Second, in Section 5, we show how to derive the major system components and their interaction behavior schematically from the captured interaction requirements.

We use the running example of Section 2, an autonomous transport system, to illustrate our approach. In Section 3 we briefly introduce the component model underlying UML-RT.

2 Running Example: Broadcasting Architecture of an Autonomous Transport System

As the running example for illustrating our methodological approach we use an autonomous transport system within a production plant. The purpose of this system is to ensure that workpieces are transferred from their present location to another where the next production step is then carried out. In the beginning, fresh workpieces reside in an "in store". Workpieces whose processing is finished are transported to an "out store". Machine tools perform the actual processing of workpieces. Whenever a machine tool is free it requests to obtain a workpiece, which is then delivered by an autonomous vehicle (termed "holonic transport system", or "HTS" for short).

Machine tools and HTSs use broadcasting to negotiate the delivery of a workpiece: a machine tool broadcasts its requests to all HTSs; the HTSs, in turn, broadcast their offer (an estimate on how long it takes them to satisfy the request). Finally, the machine tool broadcasts which HTS has "won the deal".

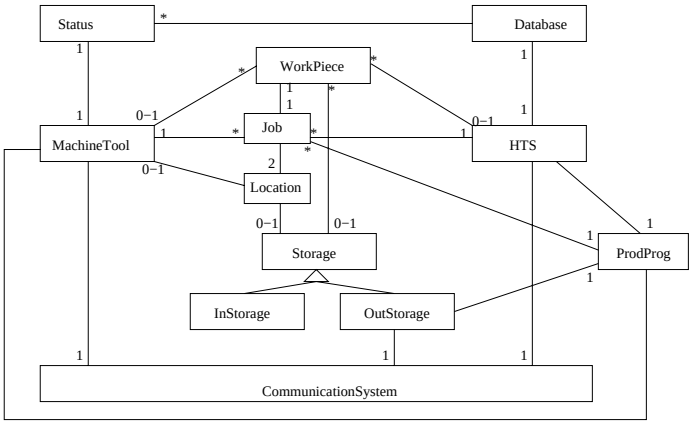


Figure 1: domain model

The domain model of Figure 1 captures the mentioned entities, as well as a few additional ones, in the form of a UML-RT class diagram. The entire production is driven

by a production plan, modeled by class ProdProg. This plan defines, among others, the required daily throughput of workpieces. The classes Database and Status model the storage of information about the HTSs' and machine tools' view of the current state of the production process. The destination of an HTS to pick up a workpiece is captured by class Location. Job is the class for modeling the pick-up tasks negotiated between machine tools and HTSs. We take class CommunicationSystem as the explicit architectural manifestation of the requirement to use broadcasting in the binary communication model of UML-RT. This domain model is the starting point for deriving an initial architecture (cf. Section 5).

3 The component model of UML-RT

In this section we give a rather dense overview of the modeling concepts of UML-RT. We refer the reader to [SR98] to obtain a more detailed understanding of the (syntactic) transfer from ROOM via UML to UML-RT.

UML-RT constitutes a merge of the ideas behind ROOM[SGW94] and the notation included in the UML[Rat98]. The key additions of UML-RT wrt. what is known from the UML are

1. hierarchic components as central elements applicable in the entire range from logical analysis to technical design and implementation,
2. a transparent non-technical notion of interfaces, defining the binary communication protocols for the interactions of components,
3. a clear communication concept: interaction between components proceeds exclusively via asynchronous signal exchange along binary communication links,
4. a clear notion of concurrency - all components are potentially active units, operating independently from all others, and
5. predefined access to the timing mechanisms of an underlying real-time operating system.

UML-RT achieves these additions essentially by means of three modeling elements: capsules, ports, and connectors. A capsule (graphically denoted by a box labeled with the capsule's name) represents a potentially active component in UML-RT whose communication with its environment proceeds by means of asynchronous signal exchange via its ports. A port (graphically denoted by a small filled or outlined square on the boundary of a capsule box) is an interface object defining the role of the capsule it belongs to within a communication protocol. Connectors (graphically denoted by a line between two port symbols) establish binary communication links between different ports, and define the protocol carried out on this link. A protocol in UML-RT consists of a set of signals sent and received along a connector. The port defined to play the role of the sender or receiver in the binary protocol is graphically represented by a filled or outlined square, respectively. The receiver role is sometimes also called the *conjugated* role wrt. the sender role of the protocol.

Capsules can nest hierarchically to arbitrary depth; an enclosing capsule communicates with its sub-capsules also via ports and connectors just as it does with its environment. There is no means for accessing sub-capsules directly from the environment of their container. The behavior of each capsule must, in particular, conform to the protocol roles the capsule commits to via its port definitions.

Consider the capsule diagram of Figure 2, which displays capsules for the HTSs, the stores, and the machine tools as an exemplary subset of the entities contained in Figure

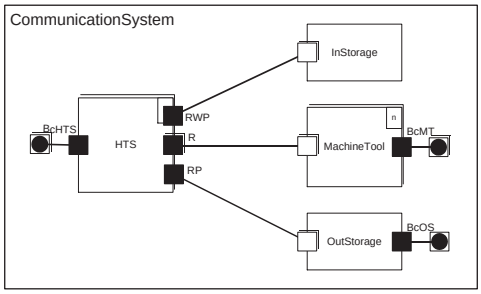


Figure 2: capsule diagram

1[1]. Each member of this set is a sub-capsule of CommunicationSystem. Every HTS has connectors to each of the stores, as well as to every machine tool, with corresponding ports. Moreover, there exist connectors between the HTSs and their container; similar connections exist for the machine tools and the out store. The ports of the container are graphically indicated by outlined squares containing filled circles.

Clearly, the restriction to binary communication protocols is a limitation especially in the context of complex technical and embedded real-time systems. In the following sections we describe how to model broadcasting communication effectively in the framework set up by UML-RT's component model.

4 Sequence Diagrams for Broadcasting

Of particular importance in defining an adequate architecture is the precise description of component interaction. UML-RT employs UML's Sequence Diagrams (SDs) in a rather loose and methodologically unfounded way. Yet, these SDs provide no notational means for dealing with broadcast communication. In this section, we show how SDs can easily be extended to model broadcast communication as well as binary communication. To discuss the extension, let us consider an application scenario of the autonomous transport system. Figure 3(a) shows the simplest case of the negotiation of a transport task.

Just as in classical SDs labeled, vertical axes represent part of the behaviour of the corresponding components. By means of labeled horizontal arrows we indicate communication via asynchronous communication. Labeled boxes denote local actions of a component. Reading the diagram from top to bottom determines an order on the interactions occurring among the components over time.

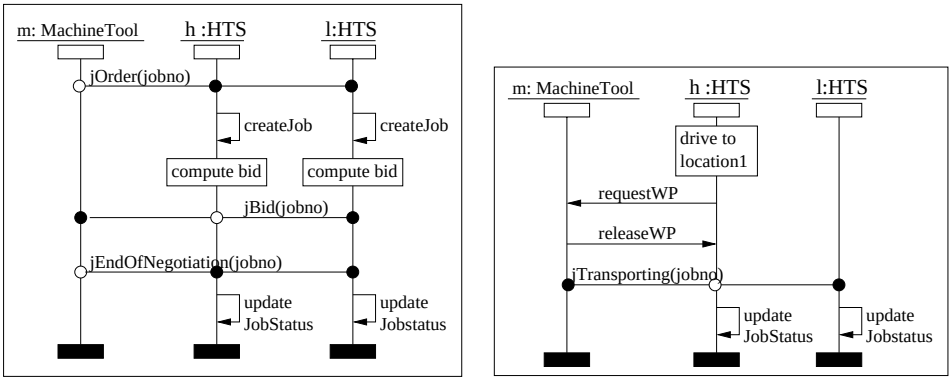
Broadcast communication is modeled by a communication line without arrow head. An outlined circle marks the originator of the message and filled circles mark the receivers of the message. This allows us also to model multicast communication. More complex scenarios, such as messages by multiple senders or iterated protocols, can be modeled using standard SD syntax such as decomposition of components or the loop construct[2]. By this notation we make explicit the presence of and the participants in the broadcast communication. We abstract from concrete implementation details, such as individual communication delays between originator and recipients of a message.

In Figure 3(a), a machine tool announces an order using broadcast communication. Each HTS stores the order in its local database which serves as a basis for the calculation of the price within the locally performed action `compute bid`. In our example scenario,

[1]We have used the syntax of ROOM which deviates only slightly from that of UML-RT, but is so far better supported by corresponding tools.

[2]Note that scenarios are interpreted as exemplary interaction patterns in the sense of [Krü00a, Krü00b]. In particular, they are not interpreted as a complete behavior specification.

only one HTS announces a bid for the order and finally, the machine tool ends the negotiation after a certain time. Figure 3(b) shows a combination of broadcast and binary communication which occurs during the execution of a transport: When the HTS arrives at a machine tool to pick up a workpiece, it sends a request to the machine tool, which responds by a release message. Finally, the HTS announces the picking up of the workpiece by means of a broadcast message.



(a) Scenario for order negotiation (b) Scenario for picking up a workpiece

Figure 3: Broadcast SDs

The semantics of the new communication construct can be easily embedded into the semantics of "normal" SDs: Each broadcast line corresponds to a set of messages, each directed from the originator to one recipient.

5 From MSCs to capsules and protocols

In this section we suggest a method for developing structure diagrams using the knowledge about our system gained during requirements analysis and expressed via the domain model and the SDs of Section 4. We show how capsules, connectors and protocols can be derived systematically and discuss the embedding of broadcast communication using these concepts. The model we obtain can serve as a starting point for the development of a system design, which can be completed, generalized and optimized by subsequent refinement steps. The advantage of the proposed procedure is that we obtain *consistency with the requirements analysis* by construction.

We start with an overview of the steps which have to be performed to get a first sketch of a structure diagram. We assume that, starting from the domain model, the active components have been identified already during domain analysis. The procedure consists of three phases: First, the capsules of the system are defined (steps 1+2, below). Second, protocols are derived from the SDs (step 3). Third, the protocols are assigned to ports which are linked by connectors (steps 4+5). The methodical steps are as follows:

1. Create a capsule for each class which appears in the SDs as an axis.
2. Create a container capsule which contains the capsules from step 1[3]. This container acts as the mediator for broadcast messages.
3. (a) Create a binary protocol for each pair of capsules which exchange messages in SDs and include all respective messages into this protocol.

[3]This step can be omitted if the container capsule is predefined already.

(b) If necessary, create an individual protocol for each capsule which uses broadcast communication.

4. Assign to each capsule its respective ports associated with the respective protocol roles.
5. Establish a connector between any two ports derived from binary communication protocols; establish a connector between any port derived for broadcasting and the container capsule.

Steps 3 through 5 are straightforward for binary communication: After protocol generation we just need to create a port for each protocol role and link the conjugated ports by connectors. Unfortunately, we cannot use connectors in such a straightforward way for broadcast communication, because in general there are more than two capsules involved[4]. We discuss two possibilities for a workaround to map broadcast communication to binary communication: The first one is to introduce a new capsule BC explicitly, which handles the broadcast communication. Therefore every capsule role which uses broadcast communication has to be connected to the role of capsule BC. As a consequence there will be a clutter of connectors. For that reason we do not follow this path further. The second one – which we use here – is that the broadcast communication is handled implicitly by the behavior of a container capsule. Each capsule which is involved in broadcast communication is equipped with a port connecting it to its container capsule. This approach has several advantages. It enables a compact way of modeling, and it also supports dealing with changing system configurations gracefully: The model need not be changed if we change the number of HTS components in the system, even dynamically.

By means of our running example we illustrate the methodological steps introduced above: We derive the capsules HTS, InStorage, OutStorage and MachineTool (step 1). These capsules are embedded into a container capsule called CommunicationSystem (step 2). For the generation of a protocol, let us consider the handshake communication HTS $\leftrightarrow$ MachineTool. From the SDs, the binary protocol `Request` (tab. 1(a)) is created. The corresponding protocol for the machine tool is easily derived by conjugation of this protocol, i.e. the exchange of send and receive messages. Analogously we proceed with other pairs of communicating capsules (step 3a). For broadcast communication we consider every

Request	
send:	requestWP
receive:	releaseWP
send:	requestPlace
receive:	releasePlace

(a)

BroadcastHTS	
send:	requestProdPrg
send:	jBid(jobno)
send:	jTransporting(jobno)
send:	jFinished(jobno)
receive:	requestProdPrg
⋮	⋮
receive:	jFinished(jobno)

(b)

Table 1: protocols

capsule and create an individual binary protocol for each capsule. These protocols contain the messages which the capsule under consideration sends and which it can receive, i.e. all broadcast messages. Table 1(b) shows the protocol `BroadcastHTS` as an example. As discussed above the ports derived to map broadcast protocols to sets of binary protocols will be connected to the container capsule which will perform the broadcast message passing (step 3b). Every capsule gets its ports associated to base/conjugated

[4] in our example there are three: HTS, MachineTool and OutStorage

roles of appropriate protocols, e.g. capsule HTS gets ports associated to the base role of Request, BroadcastHTS and other protocols which we omitted here for simplicity (step 4). Finally the connectors between the related handshake ports and between broadcast ports and container capsule are added (step 5). The result is a first prototype of the system's structure diagram. Clearly, we have to adjust the cardinality of the capsule roles HTS and MachineTool to their required number, as given in a concrete instance of the system. The resulting structure diagram is shown in Figure 2.

6 Conclusions and Outlook

In this paper, we have presented an approach which facilitates the incorporation of broadcast communication into the modeling of architectural design using UML-RT. We have shown how to integrate the notation of broadcasting into architectural modeling with UML-RT. This notation of broadcast SDs is flexible enough to model both broadcast and multicast communication and can easily be embedded into the standard semantics of SDs.

We have also introduced methodological guidelines for a schematic transformation of interaction requirements into prototypical structure diagrams. These diagrams are ideally suited to serve as a starting point for the design of the system to be developed because they guarantee consistency with the requirements analysis by construction. They can be refined in subsequent development steps: For example, new messages can be introduced or entire interaction protocols can be reorganized in order to develop more general capsule interfaces. A structuring of these development steps can be based on formal notions of refinement, even supported with guidance given by constructive rules (see for instance [Krü00a]). Furthermore, by using our SD variant, we open potential for applying fully automatic transformation techniques (such as [KGSB99]) for deriving individual component behaviour from sets of interaction patterns.

The approach of using container capsules to model broadcasting fits seamlessly with a hierarchical structuring following the Composite design pattern [GHJV95]. Therefore, it shows potential for scaling well to complex applications. The introduction of such a hierarchical mediation concept also eases the separation of different communication paradigms within a single system architecture. We refer the reader to [KPS01] for a detailed discussion of these concepts.

References

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns. Pattern-Oriented Software Architecture.* Wiley, 1996.

[DW98] Desmond D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML– The Catalysis Approach.* Addison Wesley, 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[KGSB99] Ingolf Krüger, Radu Grosu, Peter Scholz, and Manfred Broy. From MSCs to statecharts. In Franz J. Rammig, editor, *Distributed and Parallel Embedded Systems*, pages 61–71. Kluwer Academic Publishers, 1999.

[KPS01] Ingolf Krüger, Wolfgang Prenninger, and Robert Sandner. Development of an autonomous transport system using UML-RT. Technical report, Technische Universität München, to appear, 2001.

[Kru99] Philippe Kruchten. *The Rational Unified Processs. An Introduction.* Addison Wesley, 1999.

[Krü00a] Ingolf Krüger. *Distributed System Design with Message Sequence Charts.* PhD thesis, Technische Universität München, 2000.

[Krü00b] Ingolf Krüger. Notational and Methodical Issues in Forward Engineering with MSCs. In Tarja Systä, editor, *Proceedings of OOPSLA 2000 Workshop: Scenario-based round trip engineering.* Tampere University of Technology, Software Systems Laboratory, Report 20, 2000.

[Lyo98] A. Lyons. UML for Real-Time Overview. *Objectime Ltd.*, April 1998. `http://www.objectime.com/otl/technical/umlrt.html`.

[Rat98] Rational. UML Notation guide, version 1.3. January 1998.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling.* Wiley, 1994.

[SR98] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. `http://www.objectime.com/otl/technical`, April 1998.