# Deriving Architectural Prototypes for a Broadcasting System using UML-RT

Ingolf Krüger, Wolfgang Prenninger, Robert Sandner [*]

Technical University of Munich, D-80290 Munich, Germany

**Abstract**
The graphical description techniques provided by UML-RT are significant aids in modeling aspects of structure and behavior in software architectures for reactive and embedded real-time systems. Here, we sketch an extension of UML-RT's reach from binary communication to broadcasting. To that end, we introduce sequence diagrams tailored for capturing broadcasting scenarios. Furthermore, we describe systematic steps for deriving both component structure and behavior from the captured broadcasting scenarios.

## 1 Introduction

One of the key challenges in the development process for complex distributed and reactive systems is the definition of an adequate software architecture; important aspects of such an architecture are the hierarchical structuring of the system into components, the precise specification of both component interfaces and behavior, and the forces and constraints underlying the chosen decomposition.

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. Clearly, these concepts and description techniques are significant aids in developing and documenting the mentioned architectural aspects based on the binary and asynchronous communication model underlying UML-RT.

However, this binary communication model 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 in these application domains despite its lack of explicit support for broadcasting.

To address this question we sketch steps towards a seamless integration of broadcasting into UML-RT's component model; we use the running example of Section 2, an autonomous transport system, to illustrate our approach. In the subsequent sections we focus on two important methodological aspects: capturing the broadcasting requirements of the system under development (Section 3), and deriving both hierarchic component structure, and individual component behavior from the captured interaction requirements (Section 4). Section 5 contains our conclusions and an outlook.

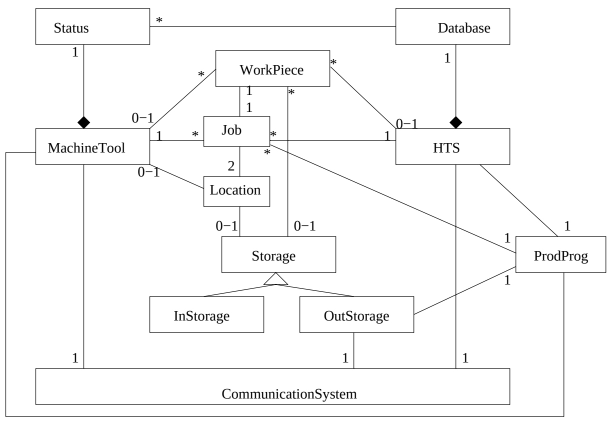
[*]Our research was supported by the DFG within the priority program "SoftSpez" (SPP 1064) under project name *InTime*.

## 2 Running Example: Broadcasting Architecture of an Autonomous Transport System

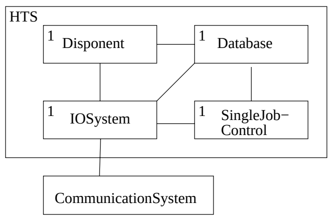
As the running example for illustrating our methodological approach we use an autonomous transport system within a production plant. The purpose of this system is to ensure that workpieces are transferred from their present location to another where the next production step is then carried out. In the 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".

The domain model of Figure 1(a) 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. Job is the class for modeling the pick-up tasks negotiated between machine tools and HTSs. The destination of an HTS to pick up a workpiece is captured by class Location. We take class CommunicationSystem as the explicit architectural manifestation of the requirement to use broadcasting in the binary commu-



(a) top level domain



(b) HTS domain

Figure 1: domain model

nication model of UML-RT.

Figure 1(b) shows the internal structure of an HTS in more detail. In addition to `Database` the HTS contains the components here captured by classes `Disponent`, `SingleJobcontrol` and `IOSystem`. Disponent and SingleJobControl are responsible for the two main tasks of an HTS: negotiating jobs, and executing an aquired job, respectively. The IOSystem handles the communication between the CommunicationSystem (for broadcasting) and the other subcomponents of the HTS. By means of this domain model we have covered the logical associations between the classes of the system under consideration. The corresponding communication paths will be identified by developing interaction scenarios (cf. Section 3). Furhtermore, the domain model is the starting point for deriving an initial architecture (cf. Section 4).

## 3 Sequence Diagrams for Broadcasting

Of particular importance in defining an adequate architecture is the precise description of component interaction. Modeling component interaction both covers important aspects of the requirements analysis and is a first design step since it identifies “active”, communicating components among the entities defined in the domain model (Fig. 1). The major modeling technique of the UML employed in this step are sequence diagrams (SDs). Yet, these SDs provide no notational means for dealing with broadcast communication. Furthermore, there is no sufficient methodological integration with other UML diagrams, such as statecharts. In this section, we show how SDs can easily be extended to model broadcast communication as well as binary communication, and to express relations to behaviour models. To discuss these extensions, let us consider an application scenario of the autonomous transport system. Figure 2(a) shows a possible scenario for 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. Rectangular 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. 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.

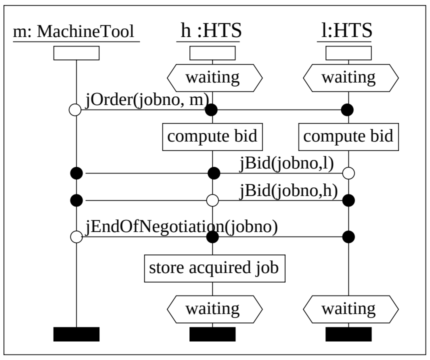
A second extension of SDs are state labels which are depicted by labeled hexagons. This notation is taken from the ITU MSC 96 specification [IT96]. State labels appear on axes in SDs; they identify control states of the corresponding component. Using state labels we can combine SDs to more complex scenarios: Different SDs starting with the same state label can express nondeterministic choice; SDs starting and ending with the same state label can indicate repetition[1].

In Figure 2(a), a machine tool announces an order using broadcast communication. Each HTS calculates how long it takes it to satisfy the request within the locally performed action `compute bid`. In our example scenario, two HTSs announce a bid for the order and finally, after the machine tool ends the negotiation, HTS h has won the deal. After the negotiation, the HTS components reside in the same state as they started. Figure 2(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.

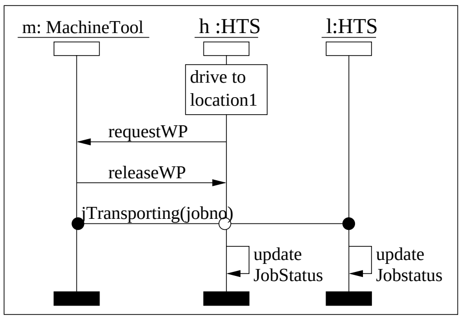
## 4 Deriving Architectural Design from Scenarios

Based on a brief introduction of UML-RT’s component

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



(a) scenario for order negotiation



(b) scenario for picking up a workpiece

Figure 2: broadcast SDs

model[2] we now discuss the systematic derivation of component structure and behavior from captured broadcasting scenarios.

**Component Structure**

A capsule (graphically denoted by a box labeled with the capsule's name) in UML-RT represents a potentially active component 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.

[2]We refer the reader to [SR98] to obtain a more detailed understanding of the (syntactic) transfer from ROOM via UML to UML-RT.

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.

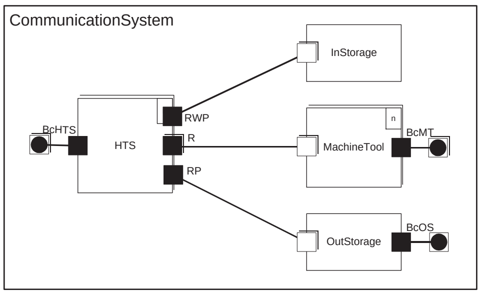


Figure 3: capsule diagram

Consider the capsule diagram of Figure 3, which displays capsules for the HTSs, the stores, and the machine tools as an exemplary subset of the entities contained in Figure 1(a)[3]. 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.

In the following 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 3. 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:

[3]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.

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[4]. 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.
   (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[5]. Instead, we handle broadcasting 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 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

[4]This step can be omitted if the container capsule is predefined already.
[5]in our example there are three: HTS, MachineTool and OutStorage

| 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)

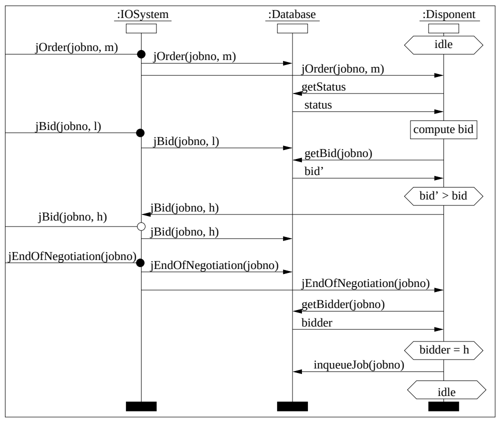
Table 1: protocols

container capsule which will perform the broadcast message passing (step 3b). Every capsule gets its ports associated to base/conjugated 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. Figure 3 shows the resulting structure diagram.

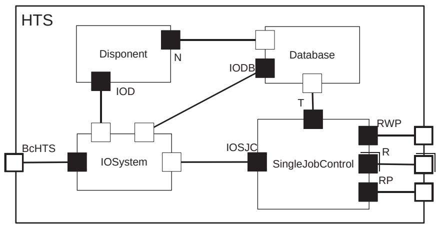
We can apply the design method presented so far also in a hierarchical decomposition of the system to be developed. We give a rough sketch of this by means of the decomposition of the HTS shown in Figure 1(b). As a first step, we refine the axis `h` in SD 2(a) with respect to its substructure, as shown in Figure 4(a). The refined SD shows the internal interactions within the HTS which are necessary to decide whether a bid is submitted for an order, and to check whether the negotiation has been successful. On the basis of Figure 4(a) and refinements of further SDs – such as Fig. 2(b) – we derive the structure diagram of the capsule HTS shown in Figure 4(b), following the scheme presented above. The ports to the environment match with ports which the capsule HTS has in its container capsule communication system (Fig. 3).

## Component Behaviour

In this section, we give a rough sketch of how to derive behaviour specifications of the system components from the scenarios collected during the requirements analysis. Again, this development step can be carried out in a schematic way, using an algorithm presented in [KGSB99]. This algorithm takes a set of sequence diagrams as input, and generates an automaton specification for the component under consideration as output. This algorithm employs the state labels introduced in Section 3 to determine execution orderings between the specified scenarios. It consists essentially of four steps: (1) *Projection*: After having selected the component for which we want to construct an automaton, we project each of the given SDs onto this component, (2) *Normalization*: We determine the transition-path segments defined by the projected SDs, according to the state labels appearing in



(a) refined scenario for order negotiation



(b) capsule diagram of HTS

Figure 4: decomposed vies of the HTS

the SDs; if necessary, we add appropriate state labels at the beginning and at the end of the projected SDs, (3) *Transformation into an automaton*: We turn every message arrow appearing in an SD into a transition of the automaton; if necessary, we add intermediate states to link transitions, such that they correspond to a sequence of messages within a normalized SD, (4) *Optimization*: We apply heuristics, or use algorithms known from automata theory for automaton minimization.

As an example, let us consider the disponent capsule appearing in Figure 4(b). The input source for the generation of a statechart of the capsule is the SD in Figure 4(a). The conditions on the axis of the disponent state that the execution ends in the same state as it started (named `idle`. Two further conditions on bids and names of bidders allow the truncation of the negotiation; they yield a split of the SD into three parts at this point in step (2) of the transformation procedure. These state conditions allow us to specify alternatives at this points which we have omitted here for simplicity. For a detailed model, we refer the reader to [KPS01]. An automaton resulting from the generation which does include choices is shown in Figure 5. Note that the local action `compute bid` appears as a state in the automaton. This allows us to preserve the structure of the automaton in further refinements of scenarios. If we specify how this action is performed, we can plug this into a substatechart of this state.

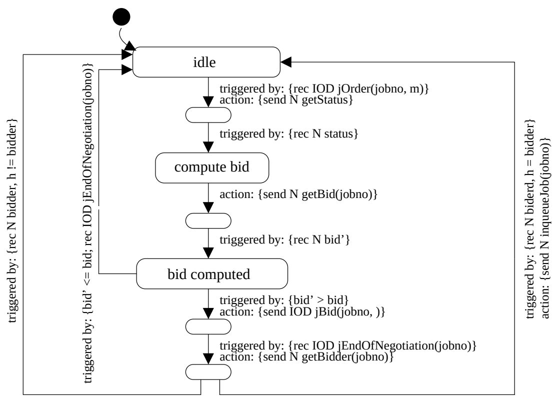

idle
triggered by: {rec IOD jOrder(jobno, m)}
action: {send N getStatus}
triggered by: {rec N status}
compute bid
action: {send N getBid(jobno)}
triggered by: {rec N bid'}
bid computed
triggered by: {bid' > bid}
action: {send IOD jBid(jobno, )}
triggered by: {rec IOD jEndOfNegotiation(jobno)}
action: {send N getBidder(jobno)}
triggered by: {rec N bidder, h != bidder}
triggered by: {bid' <= bid; rec IOD jEndOfNegotiation(jobno)}
triggered by: {rec N biderd, h = bidder}
action: {send N inqueueJob(jobno)}

Figure 5: an optimized statechart for the disponent

## 5 Conclusions and Outlook

We have presented an approach at incorporating broadcast communication into the modeling of architectural design using UML-RT. In addition, 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 shown that on the basis of a few notational elements taken from ITU MSCs, SDs can be integrated nicely with diagrams which model other important aspects of architecture: structure and behaviour. Prototypical models can be derived systematically from SDs. The resulting diagrams provide a high level architecrure description and are ideally suited to serve as a starting point for the actual design of the system to be developed, because they guarantee consistency with the requirements analysis by construction. The initial architecture 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]).

Combined with the Composite design pattern [GHJV95], our approach of using container capsules to model broadcasting shows potential for scaling well to more complex applications. We refer the reader to [KPS01] for a detailed discussion of these concepts.

## REFERENCES

[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.

[IT96] ITU-T. *ITU-T Recommendation Z.120 – Message Sequence Chart (MSC 96)*. ITU-T, Geneva, 1996.

[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.

[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`.

[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.