T U M

I N S T I T U T F Ü R I N F O R M A T I K

Development of an Autonomous Transport System using UML-RT

Ingolf Krüger, Wolfgang Prenninger, and Robert Sandner

TUM-I0215
Dezember 02

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-12-I0215-100/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Development of an Autonomous Transport System using UML-RT

Ingolf Krüger
Wolfgang Prenninger
Robert Sandner

Abstract

Capturing the system requirements and integrating them into a well balanced system architecture is a key issue in the development of reactive software systems. In this report, we explore by means of a case study how an architecture can be derived systematically for systems whose communication model is based on broadcasting. We are especially interested in two important questions: can the derivation of the architecture be automated by generation algorithms, and are the description techniques used in practice today an adequate basis for such a development process? We address the second question by applying UML-RT, a profile of the widely used Unified Modeling Language (UML) which focuses on embedded system applications, on the modeling of requirements and architecture of an autonomous transport system. Adequate graphical description techniques for capturing interaction scenarios which include broadcasting are unavailable so far. We introduce an extension to the UML's sequence diagrams (SDs) to capture broadcasting scenarios. We also address the combinations of SDs to describe complex scenarios, their hierarchical refinement, and the embedding of broadcasting into UML-RT's architectural description techniques, and discuss the specification of additional constraints within the Object Constraint Language (OCL). To support an automatic synthetization of an architecture from scenarios, we present an algorithm which generates capsule diagrams from scenarios modeled using SDs. Furthermore, we discuss the adaption of an existing algorithm which generates statecharts from MSCs to fit with their dialects used in UML-RT, namely UML-RT statecharts and sequence diagrams.

Contents

1 Introduction

Capturing the system requirements during analysis and integrating the various competing objectives into a well balanced system architecture is one of the decisive tasks in the development of complex software systems. While this is already a challenging task for arbitrary systems, embedded real-time systems typically pose additional problems: strict resource limitations, such as timing and memory constraints, complement the usual functional requirements aspects; moreover embedded real-time software architectures are often intertwined or constrained by the underlying infrastructure, such as real-time operating systems and their specific scheduling and communication paradigms.

Given the importance of deriving an adequate software architecture from the captured requirements, two key challenges arise:

- how to document system requirements and architectural decisions in a precise, yet transparent way such that the important ideas can be easily communicated to the participants in the development process, and
- 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?

For the description part, UML-RT [SR98, Lyo98], a sequel to ROOM [SGW94], has been suggested as a notation for representing both important system requirements, namely interaction scenarios and their constraints; and for modeling architectural aspects, such as hierarchical decomposition of components, communications relationships and interfaces, and individual component behavior[1]. The corresponding description techniques are sequence diagrams, the object constraint language, capsule (and class) diagrams, and a subset of the UML's statecharts. These are significant aids in capturing both requirements and important architectural aspects.

Mastering the second challenge can be addressed by both methodological and algorithmical support for the mapping of requirements into artifacts representing parts of the system's architecture. Scenarios elaborated in the course of the analysis mostly identify central components of the system to be developed, and provide an elementary behavior specification for them. The algorithm presented in [KGSB99, Krü00a] uses this information for a fully automatic generation of statecharts describing this behavior. These statecharts serve as an ideal starting point for the design process; thus, analysis and design can be seamlessly integrated.

[1] UML-RT is a set of notations extending and complementing the standard UML by concepts of ROOM for the modeling of architecture and time. It is a major contribution for an improvement of the UML standard in its version 2.0 and a *UML profile for scheduling, performance, and time*. Both are currently standardized by the OMG.

In this report, we assess the potential of applying UML-RT's requirements and architecture modeling capabilities and of algorithms for the generation of prototypical architectures by means of a case study. The application developed within the study, a holonic transport system[2] within an automated production plant, represents a typical application domain for embedded systems[3]. To model requirements and architectural aspects appropriately, we identify extensions of UML-RT for a complete coverage of the application domain of embedded systems, and structuring mechanisms for models. To establish a seamless integration of analysis and design, we adapt the generation of statecharts from scenarios [KGSB99, Krü00a] to the specifics of UML-RT, and we present an algorithm which also enables the derivation of the static part of architecture descriptions, i.e. capsules, ports, protocols and connectors, in a systematic manner.

The work underlying this report is part of the research project *InTime*[4] which develops both a methodological and semantical foundation of the core concepts of UML-RT. The goal of this effort is to support a stepwise, incremental development process for embedded software systems with real-time constraints. Based on the foundation of UML-RT, in *InTime* we develop methods for propagating requirements into a system architecture, and define refinement rules which allow for transforming an architecture into an implementation by correctness-preserving, manageable development steps. Together, these techniques and methodological approaches provide a framework for a rigorous development process from requirements to an abstract implementation, yet based on practically proven description techniques. This process is shortly described in the next section.

Outline: The report is organized as follows: Section 2 provides a dense overview on the ideas which underly the development process adopted in *InTime*. It also sketches the key concepts of UML-RT. Section 3 gives an informal description of the subject of the case study, the transport system. Section 4 provides an analysis and design model of the system, which is refined at more detailed abstraction levels in Sections 6, and 7. Section 3 also contains a principal discussion of the generation of structural models from the scenarios discovered during analysis. Section 8 discusses the generation of behavior models for the system components, and finally, Section 9 summarizes the results of the study and contains concluding remarks.

[2]The term *holonic transport system* is used in the automation domain to denote automated, autonomously operating vehicles carrying out the transport task.

[3]The study originates from the priority program "SoftSpez" funded by the Deutsche Forschungsgemeinschaft and serves as a reference application to compare various modeling approaches for embedded software systems [fPuAI99].

[4]Funded by the Deutsche Forschungsgemeinschaft within the priority program "SoftSpez".

2 A Brief Overview on the Development Process

UML-RT brings dearly needed modeling concepts for aspects of software architecture and real-time into the UML. Being a UML profile, it also inherits all notations of the standard UML. The application area of many diagrams of the UML is not defined clearly; some diagrams partially overlap with others in scope (as an example, consider statecharts, sequence and collaboration diagrams). In this section, we give an overview of the modeling techniques we use in our approach and shortly describe their application in the development process. The process is centered around the architectural modeling concepts originating from UML-RT's ancestor ROOM which we briefly recall in Section 2.1. Figure 1 gives an overview on the interrelation of the modeling techniques used within the analysis and design phase of development.

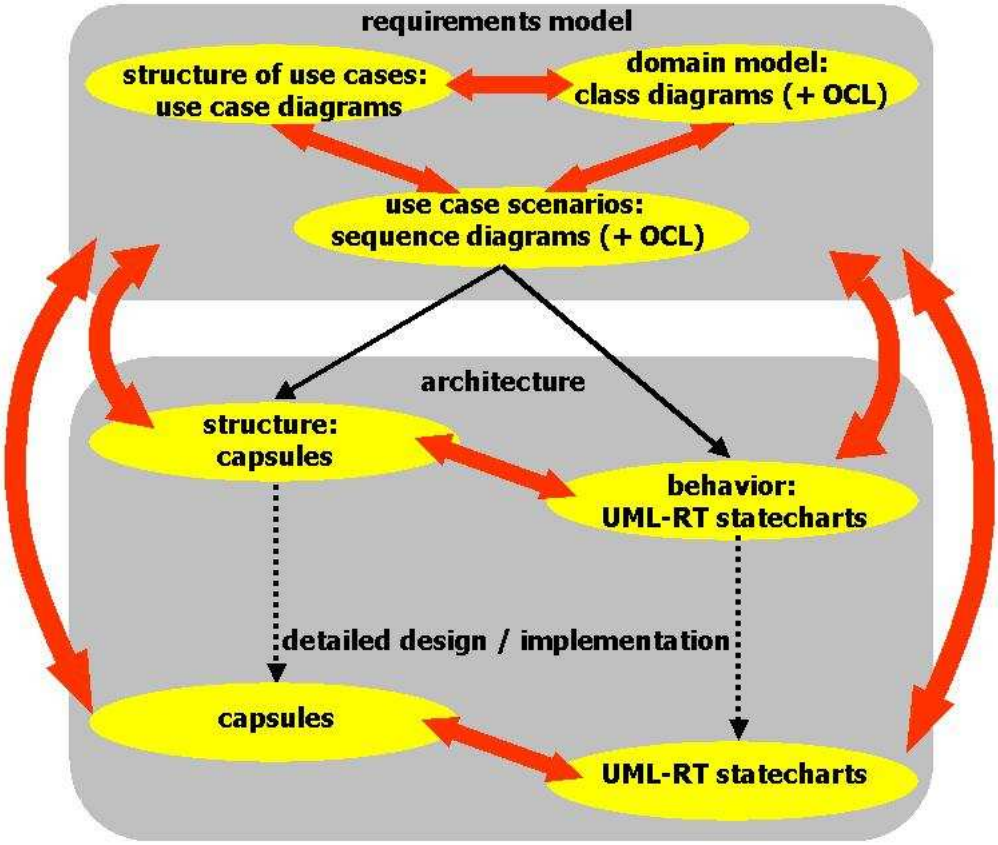


Figure 1: UML-RT techniques in the development process

The development process starts with a requirements model of the system to be developed. Following prominent methods for object oriented analysis, this phase is driven by the analysis of use cases in our approach.

The resulting requirements model is described by three interdependent diagrams: Dependencies among the use cases and their relation to major system components and the environment are modeled using *use case diagrams*.

Exemplary interaction scenarios which appear in the course of a use case between the system and its environment or between system components are specified by *sequence diagrams*. Usually, only the most important of the possible interaction scenarios for a use case are specified in this phase. Less important scenarios, e.g. handling of specific errors or exceptions, is usually not included at this stage of the development process.

The specifications in use cases diagrams and sequence diagrams depend on a *domain model*, described by a *class diagram*, which clearly defines users, components (as regarded in the analysis) and application concepts like information interchanged in the course of interactions.

Both sequence diagrams and class diagrams are not expressive enough to formalize every constraint which may be identified in the course of the analysis adequately[5]. For this reason, they may be supplemented by constraints expressed in the object constraint language (OCL). In general, these diagrams are not developed in a predefined sequence but, due to their various dependencies, in an iterative process.

When the requirements model has evolved sufficiently mature, the process is extended for a development of an initial architecture of the system, which is the starting point of the design process. We describe the structural part of the system architecture by *capsule diagrams* and the behavior of the system components - called capsules - by *UML-RT statechart diagrams*. To achieve a seamless integration of requirements and architectural models, we generate a prototypical architecture directly from the requirements model, using the algorithm presented in [KGSB99] and a second algorithm presented in Section 5. All the information needed for this generation is included in the interaction scenarios described by the sequence diagrams.

Again, the development of the architecture, and the subsequent detailed design, is an iterative process which includes the requirements model. These interdependencies are illustrated by the two-way arrows in Figure 1, whereas the black unidirectional arrows show the primary information sources for the generation and refinement steps. Usually first the structural part of the prototypical architecture is generated. In incremental development steps this structure is refined, e.g. by breaking up the generated capsules into smaller units, or by generalizing component interfaces. These changes require an update of the scenarios in the requirements model in order to reflect the refinements, which may also lead to a more detailed requirements specification.

The generation of behavior descriptions starts after basic components which are not broken up further have been identified. Based on (refined) interaction scenarios, statechart specifications are generated by the algo-

[5]For example, conditions for optional associations in class diagrams.

rithm presented in [KGSB99]. Again, these statecharts are refined in a stepwise manner, influencing the system structure and also the requirements specifications. Yet, these mostly lead to less fundamental changes in the models developed before.

To summarize, both the construction of the system and its requirements are developed in an iterative, stepwise manner involving numerous refinement steps. This parallelism in the development process is illustrated in Figure 2.

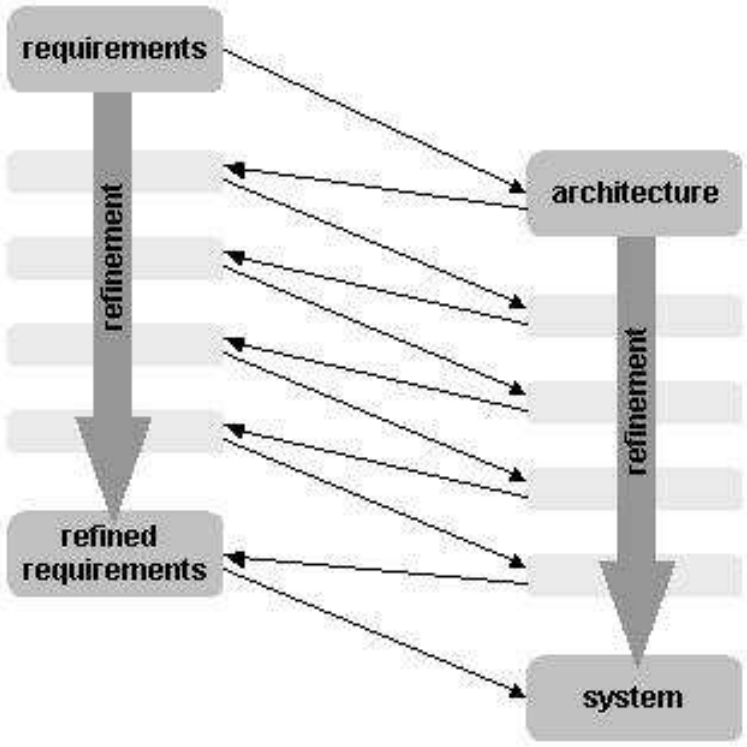


Figure 2: Interactive refinement of requirements and system architecture

Developing a system based on the refinement of a prototype directly generated from the requirements model seems to break with many traditional approaches and to be fairly constraining at a first glimpse. In fact, it requires a very strict development process. However, the tight coupling of requirements analysis and architectural design does not prevent developers from correcting inappropriate system structures used in the analysis nor does it enforce to deal with design matters already during the analysis. On the contrary, changes in the system architecture are part of the presented development process. Enforcing that changes in the system architecture are also properly reflected in the requirements specification prohibits that important requirements get lost during restructuring of the model. This is an important step towards a seamless chain in the development process from the requirements up to an implementation, which allows to keep

track of the fulfillment of each requirement imposed on the system. Thus, the integration of automatic generation, which establishes correctness by construction, and of refinement steps is an important contribution for the development of dependable systems.

2.1 Essential modeling principles of UML-RT

In this section, we give a short introduction to UML-RT's core modeling concepts. We discuss its major structuring principle, the notion of components, called capsules, and important principles of interaction and behavior models. We omit a thorough syntactic introduction of UML-RT's rich modeling language here; if the reader is not familiar with its notations, we recommend [RS01] for a short introduction, and [SR98, Lyo98] for a complete reference.

A major contribution of UML-RT to the standard UML is the addition of concepts for modeling system architectures. Core principles of UML-RT include

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 the interfaces of two 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 each 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 adding three modeling elements to the UML: capsules, ports, and connectors. A capsule represents a potentially active component in UML-RT whose communication with its environment proceeds by means of asynchronous signal exchange via its ports. Each capsule is equipped with a FIFO queue to store messages sent to it until the capsule is ready to process them; and the sender of the message is not blocked until it gets a response.

A port is an interface object defining the role of the capsule it belongs to within a communication protocol. Connectors establish the 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 a 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 itself to via its port definitions.

Sequential statecharts, i.e. statecharts without parallel or AND-states, represent the behavior of individual capsules. Like capsules, states may nest to arbitrary depth; each state can enclose its own (sub) statemachine. The syntax of UML-RT's version of statecharts is similar to the one of the standard UML; one difference is that in UML-RT there is a strict separation of hierarchy levels: control flow between a state and its sub-statechart can only be passed via so called transition points. Naturally, the semantics of UML-RT's statemachines differs from the standard UML: Along the lines of the asynchronous communication model, the UML-RT's statecharts base on an asynchronous execution model, allowing the statemachine of a capsule an arbitrary delay in the execution of an action.

3 An informal description of the system

In this section we describe informally the underlying case study of this technical report. The whole system has the task to deburr a number of workpieces. The system consists of the following devices:

- three *machine tools* which deburr and wash workpieces, respectively,
- an *in storage* which provides untreated workpieces,
- an *out storage* which receives the completed workpieces,
- a holonic transport system consisting of three autonomous working vehicles - called *holons* - which carry out the transport of workpieces between the machine tools and storages.

Each workpiece processed by the system is taken from the in storage, treated by the first, second, and third machine tool in this order and finally delivered to the out storage. We call this a rigid flow of material. The machine tools have in addition to their workplace a buffer for two untreated workpieces.

The transport of the workpieces between the storage units and the machine tools is organized by means of negotiations which use broadcast messages: One machine tool sends a broadcast message when it has finished one workpiece which has to be carried away. The holons respond individually to to this order by sending an offer containing the cost of the transport (i.e. the time it needs to carry out the transport). In that sense, they have authority to decide over themselves and over the system.

Each holon has a buffer to carry one workpiece. Furthermore, every holon has an internal database which contains a complete copy of the working process. They use the database to make their decisions and keep it up to date with the received broadcast messages.

In the following we briefly list major scenarios of this production system:

Initialization of the production:

1. One of the holons asks the out storage about the work plan, i.e. how many workpieces should be treated.
2. The out storage sends the work plan via broadcast.
3. A machine tool posts a job and sends its status via broadcast and the production starts.

Negotiation of jobs between a machine tool and the holons:

1. A machine tool posts a job to carry away a treated workpiece. Simultaneously it sends its status.
2. The holons receive the message and update their database. They start to compute a bid.

3. One holon sends its bid. The other holons listen and send only a bid, if they can under-bid it. If no holon sends a bid, then the machine tool will post the job again after a certain time.
4. After a fixed time the machine tool ends the negotiation and the holon with the lowest bid receives the job.

Transfer of a workpiece:

1. a) A holon requests a workpiece from the in storage or the machine tool, respectively.
   b) A holon requests place from the out storage.
2. a) The Machine tool or the in storage releases the workpiece, respectively.
   b) The Out storage releases a place.
3. The holon acknowledges the successful workpiece transfer via broadcast.

4 A Requirements Model of the Transport System

As mentioned in Section 2, in a first step, we explore the tasks of the system we are going to develop. From the description given above, we've got a rough idea of the physical components of the system: We have an arbitrary number of work pieces which are processed by three machine tools, transported by three holons and stored in two storage facilities. Our task is to develop the control software establishing the work piece transport. The textual scenarios in Section 3 describe how the transport is organized, albeit in an informal manner. Thus, as a starting point of the development process we have to make this information more precise and complete.

First, we identify single tasks the holonic transport system (short: HTS) have to perform, i.e. we develop a use case model. As a second step, we give a more formal model of the system structure, where we in particular turn our attention to the information necessary to control the work piece transport. Based on this model, we can model scenarios which describe the information flow for each use case using Sequence Diagrams. Each model is developed in an iterative manner. Also, some alternative approaches will be addressed.

4.1 Use Cases of the System: A Use Case Diagram

Let us start with a first structuring of the tasks mentioned in the informal description. The holonic transport system is responsible that a daily stint of work pieces (defined by the production program) will be produced in cooperation with the storage units and the machine tools. The informal description given in Section 3 also identifies a number of sub tasks such as negotiation of a job and the transport of a work piece. We reflect this structuring in the use case diagram shown in Figure 3:

First, the diagram constrains the system to be developed as the HTS transport system. The tasks identified above are modeled as use cases of the system and interrelated using `<<includes>>` relations the according to their structure. We also constrained the possible interacting components outside the system for each use case. The diagram also shows two separate use cases which descend from the informal specification: The handling of errors and moves of the HTS towards more advantageous locations during idle time. Although any of the mentioned use cases can appear during the fulfillment of the production program, the last two are not considered as sub use cases of the `organize production program` use case since they (at least the errors) are not intended to occur within it's execution. The diagram shown above is not necessarily complete: it may be extended, for example by extending the use case maintenance. We dispensed them because we restrict our attention to the described informal specification

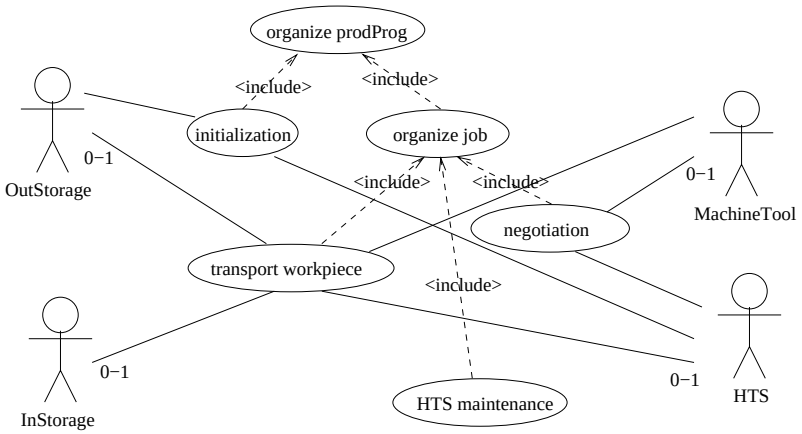


Figure 3: Use cases taken from the informal specification

given in [fPuAI99].

4.2 Objects in the Domain: A Class Diagram

Now we turn our attention to a structuring of the objects of the system. Let us start with a quite simple model which just models the known physical components of the system as classes and shows their interrelations:

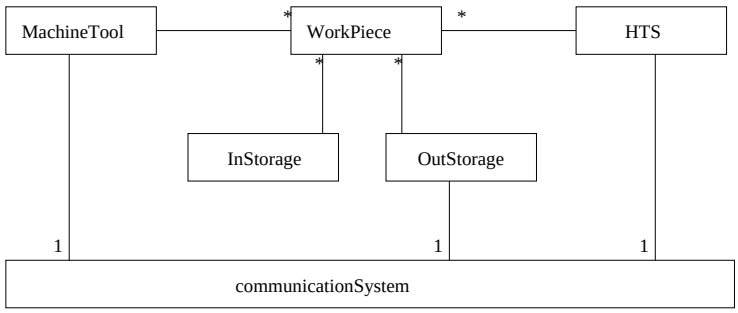


Figure 4: A simple class model

Clearly, the work pieces are associated to all physical units of the system. An important aspect in the informal specification in the previous section is that different communication paradigms are used in the system: Mainly broadcast communication is used, but some local tasks are also negotiated using peer communication. Thus, the communication system seems to be a central part of the system and is therefore explicitly modeled as a component. This model also allows us to keep the class diagram independent of communication specifics but allows to defer that issue for later, when the component `CommunicationSystem` is considered. For the development of the transport system, we need to extend our model for the information relevant for the planning. Notions mentioned in the informal

description which have to be incorporated into the domain model consist of e.g. a machine status or the daily production program. The class diagram in Figure 5 extends the model shown above, and does slightly restructure the classes:

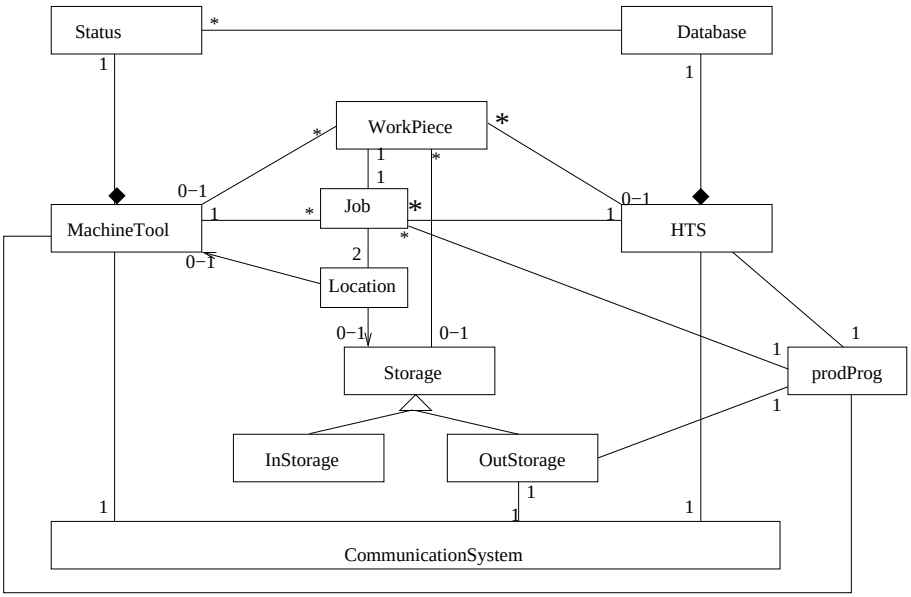


Figure 5: A more comprehensive model

Work piece transports are modeled by the class `Job`. Each job is posted by *one* machine tool, negotiated with (a set of) HTS components and finally delegated and executed by an HTS. Thus, each job is associated with one machine tool but - due to the negotiation - with arbitrary many HTS, and each machine tool and HTS is related to arbitrary many jobs. Further, a job is associated with two `Location`s which model the source and destination machine/storage unit from/to a work piece is being transported[6]. These buffers are associated with machine tools or storage units.

`InStorage` and `OutStorage` have analogous capabilities; this is expressed by a generalization relationship in 5 but `OutStorage` has some additional relationships.

As mentioned in the informal specification, the whole process is initialized by the announcement of the daily production program `ProdProg` by the out storage unit. This program involves a number of jobs and must be known by all HTS.

Finally, we include the information in the model that the major components maintain their own local data: Each machine tool has a *status*, and each HTS maintains its own database in which it stores all necessary process information to negotiate and carry out jobs.

[6]We assume that each machine tool has only one interface for the delivery and removal of work pieces and that organizes its buffers, e.g. by pallets.

A note should be made concerning the multiplicities specified for the associations: Multiplicities are specified where they are implied by the informal specification and seriously affect the development of models for the components of the system. They only model *constraints* on the system to be developed but neither *responsibilities*, i.e. which component has to ensure that the constraints are met, nor ownerships. The latter can be expressed by arcs at one end of an association. We specified arcs for the association from class `Location` to the storage and machine tool class since this can be specified at this stage of development without being subject to change later on. We also left several multiplicities open: E.g. the number of machine tools or HTS components is left open because adding or removing components should of course always be possible. Note that this differs from the multiplicity "$*$". Whereas the latter expresses that we intend to allow an association of arbitrary multiplicity, we defer the decision to later development steps in the case of unspecified multiplicities.

Constraints There are still a number of sensible constraints which cannot be expressed by the class diagram in Figure 5. They have to be expressed using OCL:

- **context** `l: Location` **inv**:
  `l.machineTool->size + l.inStorage->size = 1`
  As stated above, each location models a source or destination buffer of a job. Thus, it may only refer to either a buffer of a machine tool, the in storage or the out storage.
- **context** `w: WorkPiece` **inv**:
  `w.machineTool->size + w.hTS->size + w.inStorage->size = 1`
  Similar: Each work piece has only one association to one of the other physical units each time.
- **context** `h: HTS` **inv**:
  `h.prodProg = h.communicationSystem.outStorage.prodProg`[7]
  All associations to a *daily plan* refer to the same plan[8].

In this domain model, we focused on the identification of classes and their associations. Thus, constraints in domain models primarily affect dependencies between those associations. Classes and associations make up the core of OCL's navigation mechanism; the language provides a sufficient set of predicates and functions to express all constraints we identified for the domain model shown in Figure 5. However, the mechanism to access every element in a formula by navigation on classes and associations sometimes leads to awkward formulas for simple formalizations: the constraint

[7]The '='-operator is defined for the type OclAny and hence for all types of the model which are subtypes by definition. The operator evaluates to True, if the both objects are the same. (see also UML 1.3 p. 7-28)

[8]Some authors use the stereotype `<<singleton>>` to express that there is a single instance of a class, e.g. [Dou98].

`h.prodProg = h.communicationSystem.outStorage.prodProg` (and analogous constraints for `MachineTool` and `Job` simply express that at any time, there exists only one instance of class `h.prodProg`. Operators which quantify on object instances instead of classes may be helpful here.

Abstraction levels for domain models The amount of information we formalized in the domain model discussed above may seem arbitrary. Based on the informal description which was the starting point of our model, we can draw a model which is still finer grained. Consider, for example, the extended model shown in Figure 6 :

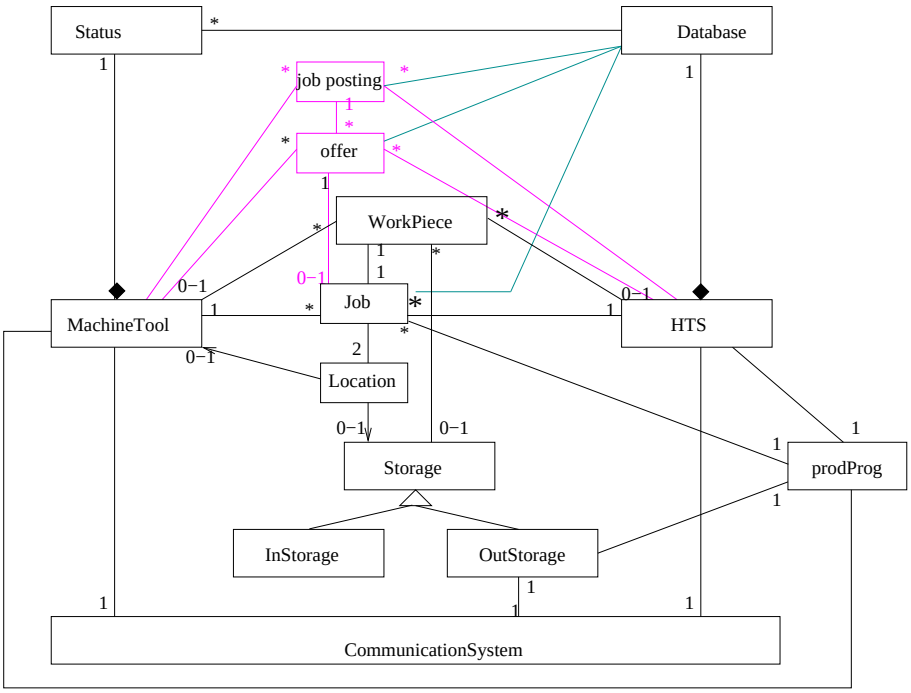


Figure 6: Extension to the domain model

The first extension includes various objects involved in the life cycle of a job: Jobs are advertised for bids by *job postings*, and the HTS may respond with *offer*s. The concept of the *job* is still present in the model. It means a job that is already delegated to and maybe processed by an HTS. We have the following relations: A job posting involves zero or more offers. For a job, there may have been many offers, but an offer may only lead to one job. Each job is now related to (delegated/carried out) exactly one HTS. On the other hand, an HTS can be responsible for an arbitrary number of jobs each time[9]. The other relations involving, job postings, offers and jobs are self explanatory.

[9]While proceeding a job a HTS can already negotiate and accept new jobs for optimal utilization.

The second extension of the model makes explicit that the classes *job*, *offer* and *job posting* are, though they are artifacts which exist independently of their representation, stored within the local databases of the HTS.

Whether these extensions should be included in the domain model, partly depends on the extent to which we would like to constrain implementations of the system. Including the classes *offer* and *job posting* in the model would lead us to an implementation which stores the information on a job also in three different classes. It would also include information on the negotiation process of a transport task like *a job is posted and this posting is responded by offers, but the HTS do not ask for jobs.* Though this is intended by the informal description, however, the information dealing with the negotiation *process* should be modeled in sequence and behavior models but not in the domain model. Therefore, we do not consider the extensions of Figure 6 further and stick with the domain model shown in Figure 5.

4.3 Description of the Use Cases: Sequence Diagrams

From the domain model described above, we have got a clearer understanding of the notions of HTS, machine tools, jobs etc. Thus, we can develop a precise description of what's going on in the course of the Use Cases identified in Section 4.1. We achieve this by specifying interaction scenarios for each use case within sequence diagrams (SDs). In the remainder of this section, we first discuss how to model the communication concept of broadcasting within SDs, and develop scenarios for the use cases identified in Section 4.1.

4.3.1 Modeling Broadcasting: Broadcast SDs

The UML's major modeling technique employed in the requirements analysis to model interaction scenarios are sequence diagrams. While this description technique works fine for binary communication, almost surprisingly, there exists no notational means for dealing with broadcast communication. In the following, we show how SDs can easily be extended to model broadcast communication as well as binary communication, and to express relations to behavior models. To discuss these extensions, let us consider an application of the autonomous transport system. Figure 7 shows the scenario for the negotiation of a transport task.

The syntax of the diagram is the same as for classical SDs: vertical axes – called life lines – represent part of the behavior of the corresponding components which are represented by the labeled boxes on top of the life line. Labeled horizontal arrows indicate asynchronous communication. Rectangular labeled boxes in the life line denote local actions of a compo-

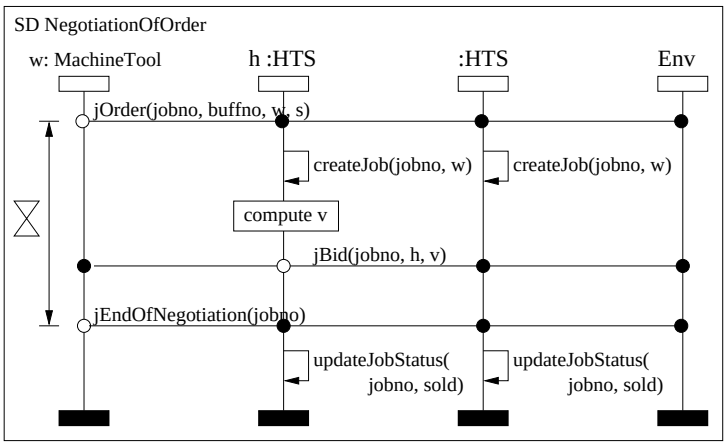


Figure 7: Broadcasting within sequence diagrams

nent. Reading the diagram from top to bottom determines the order of the interactions occurring among the components over time.

Broadcast communication is expressed by communication lines without arrow heads. An outlined circle marks the originator of the message and filled circles mark the receivers of the message. This models broadcast and even multicast communication succinctly. We call this extension *broadcast SDs* in the following.

Broadcast SDs enormously reduce the complexity of scenarios: If we used classical SDs, we would have to model a broadcast component explicitly, and draw a separate arrow from this component to each recipient of the message. The resulting models are many times larger than models using broadcast SDs, on the cost of loss of intuition. Broadcast SDs can be understood as an abbreviation of these sequence diagrams. The semantics of the new communication construct is 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 life lines 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 express nondeterministic choice; one SD starting and ending with the same state label indicates repetition. Both simple and combined scenarios should not be understood as complete behavior specifications. They are interpreted as exemplary interaction patterns in the sense of [Krü00a, Krü00b].

4.3.2 Developing Scenarios for the Transport System

Let us develop scenarios for the identified use cases now. The actions of the "abstract" use cases `organize ProdProg` and `organize job` mainly consist of the actions involved in the more detailed use cases `initialization`, `negotiation` and `transport workpiece`, which have been mentioned in the informal description. How these use cases are interrelated with respect to their order of execution, i.e. the specification of condition labels, is considered later. At the moment, let us concentrate on the interactions involved in these use cases themselves.

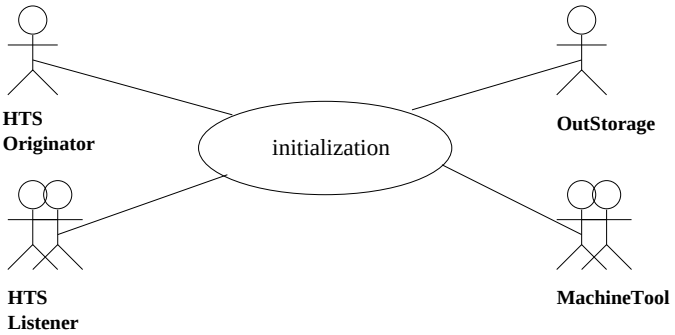


Figure 8: Initialization of the production process

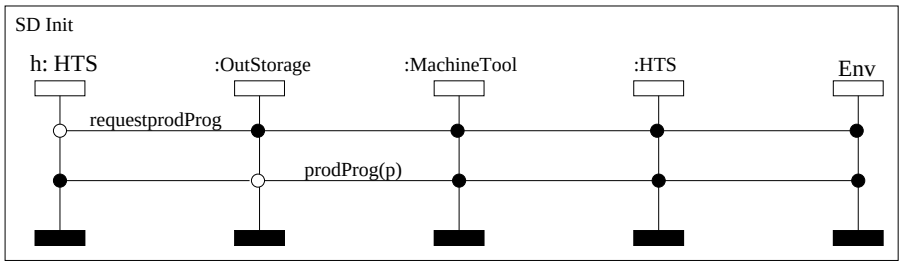


Figure 9: Initialization of the production process

We start with the use case `initialization`: From the informal description we know that *some* HTS asks the out storage for the amount of workpieces to be produced. Thus, instances of the class HTS play two roles in this use case: One HTS which asks for the production program and pure listeners. Therefore, we refine the use case diagram for the initialization in Figure 8. The sequence diagram (Fig. 9) describing the involved interactions is simple: One instance of class HTS sends a broadcast message asking for the production program and the instance of the class out storage responds to it, again with a broadcast.

For the use case `negotiation` we apply the same refinement of introducing two roles of the class HTS, as shown in Figure 10. An instance of

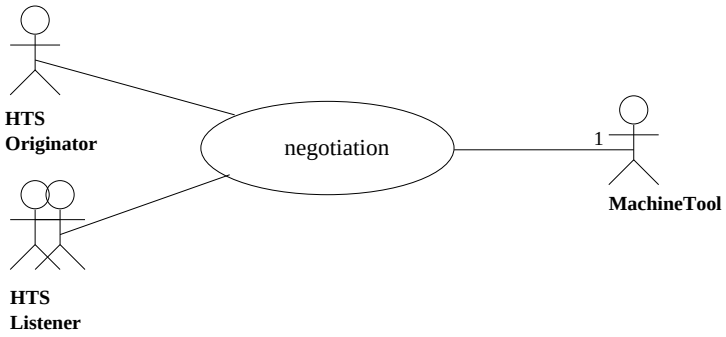


Figure 10: Use case negotiation of order

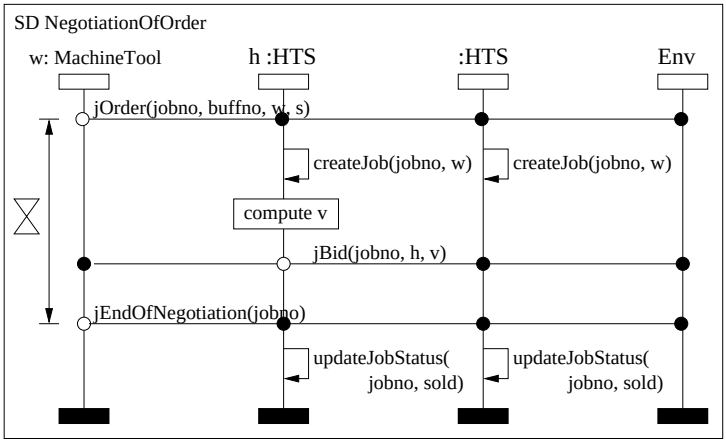


Figure 11: Sequence diagram of use case negotiation of order

class MachineTool announces a transport order using broadcast communication. A HTS decides to enter the negotiation process for this order, computes a price for the transport and sends its bid via broadcast. After a predefined delay, the machine tool ends the negotiation. This example shows that our syntactical extension of sequence diagrams is not only suitable for the specification of "pure" broadcasting but also for the specification of more elaborate communication scenarios: additional constraints like timing information can be represented by standard SD syntax, as shown in Figure 11.

Of course, this is only one scenario among many others: Any HTS may respond to the order, they may respond interleavingly. Modeling all these scenarios using sequence diagrams would be exhausting. Our goal for the moment is to understand the essential interactions of the negotiation. We defer the complete specification until we specify the behavior of the classes using statecharts.

The use case `transport workpiece` shown in Figure 3 consists of the two use cases `HTS_takes_WP` and `HTS_releases_WP`, as shown in Figure 12. Further, for these use cases we impose two constraints which can-

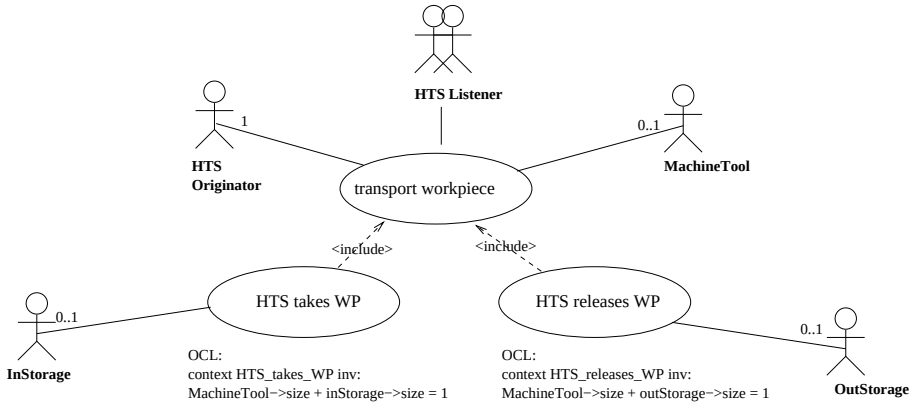


Figure 12: Use case transport workpiece

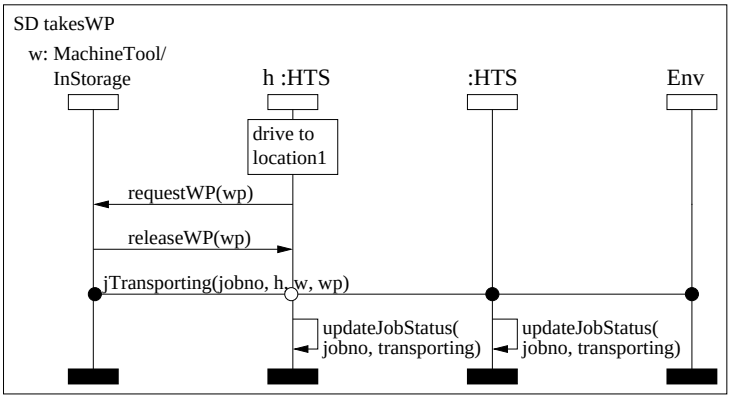


Figure 13: Scenario of HTS takes workpiece

not be expressed in use case diagrams, using OCL: Either a machine tool or a storage unit can be involved in the use cases, but exactly one of them for each instance of the use case. Note that in the literature OCL constraints are only used in relation with class diagrams and operations defined in class diagrams and not together with other diagram types. Here we use invariants in context of use case diagrams to constraint its instances (namely its associated scenarios). The sequence diagram describing the use case `HTS_takes_workpiece` in Figure 13 involves an action `drive_to_location1`. We do already anticipate that the HTS may perform actions in advance of an order, but do not specify any details at this moment. The rest of the scenario is simple: The HTS requests the release of the workpiece using directed communication, the machine tool answers, and finally the HTS announces via broadcast communication that it is carrying out the job now. This message is registered by all HTS. The sequence diagram in Figure 14 describes the release use case `HTS_releases_workpiece` (including the action`drive_to_location2` for a later specification of the transport) which is quite similar.

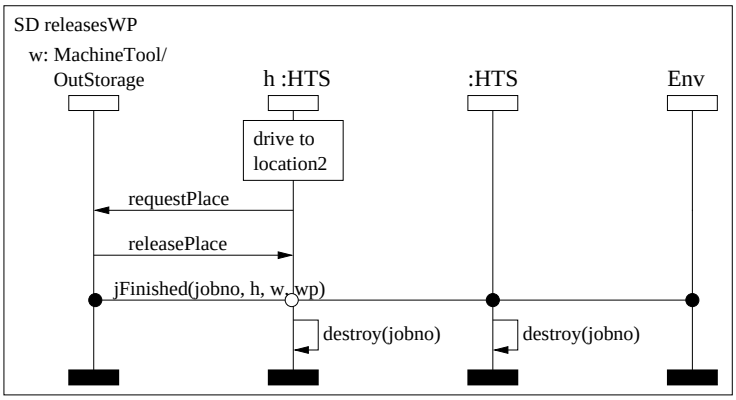


Figure 14: Scenario of HTS releases workpiece

Constraints Although the sequence diagrams presented in this section provide us with an intuitive description of the scenarios, there remain questions if we look at them more closely. Most messages in the SDs have their parameters, and the values of these are not specified clearly. As an example, let us consider some parameters in Figure 11: the message `jOrder(jobno, buffno, w, s)` has four parameters. The type (or range) of their values is not specified so far. In particular, it is not clear whether these values can be chosen freely for each message, or whether they are constrained in the context of the SD. For example, the parameter `w` should be the name of the machine tool which originates the message. Also, the parameters `jobno` should be the same in all messages. This constraints seem to be straightforward; yet, a missing definition of the parameters of the SD can lead to ambiguities: For example, does `compute v` mean that `v` is a parameter? Also, some parameter are intended to be fixed, e.g. `sold` in the internal messages of the HTS components.

To avoid such ambiguities, a formal datatype definition language is necessary which can be used if SDs are intended provide a precise specification. The OCL's datatype primitives would be sufficient to specify the types of the parameters used in our diagrams. However, again there is no possibility to relate such definitions with elements of the notation of sequence diagrams, neither to interpret an SD within the context of a comprehensive datatype definition, nor to define constraints on values which apply only to a particular SD (or elements of an SD).

Nevertheless we remove easily some of the ambiguities in the SD described above by applying the following simple convention to all SDs: In the scope of a SD identical parameter names or object names have the same value. For example, in the scope of SD NegotiationOfOrder (Fig. 11) the parameter name `jobno` has the same value in all messages like jOrder, createJob, jBid etc. Similarly, the object name and the parameter name `w`, respectively, have the value of the pointer to the instance of the

class MachineTool represented by axis labeled with `w:MachineTool`.[10]

5 Describing the System Architecture

5.1 Modeling Broadcasting in Capsule Diagrams

As we have introduced an extension of the UML's sequence diagrams to model broadcasting appropriately in the requirements analysis, we need to determine which representation of broadcasting is adequate when we define the system's architecture. In UML-RT, the essential aspects of architecture - components, communication relations and interfaces - are modeled with capsules and capsule diagrams, respectively. We face several options to include notions for broadcasting in capsule diagrams.

We could, for instance, select a broadcasting-based execution model as provided by standard statecharts[HP98], and avoid the binary communication model of UML-RT right away within the requirements analysis and architecture descriptions. As a consequence we would, however, commit to a very design- and implementation-oriented description technique and an execution model at a very early stage in the development process; in particular, there would be no clear separation of structure and behavior in the system decomposition. In addition to inheriting all of the other problems statecharts bring along with respect to their semantics (cf. [Bee94] for an overview), we would lose much potential for systematic abstraction and refinement of individual components; this is due to the lack of clear component interfaces in statecharts.

Another approach which seems to be useful at a first glance is to base the modeling of broadcast communication on a specialized version of UML-RT's ports, namely *service provision points* (SPPs), and their counterparts, *service access points* (SAPs). Indeed, SPPs are one particular way of implementing broadcasting within the current UML-RT toolset. Adding a broadcast SPP to UML-RT's virtual machine which can be used by capsules to send and receive broadcast messages would lead to a compact representation of this communication paradigm. However, the use of SPPs and SAPs hides the actual communication structure. Furthermore, more flexible communication regimes than pure broadcasting would lead to a proliferation of SPPs and SAPs; this, in turn, would contribute to obscuring as opposed to clarifying the system's architecture[11]. Also, a component owning an SAP cannot relay it to subcomponents. This is a severe drawback since this breaks UML-RT's compositional refinement mechanisms; a component cannot be refined further without considering its environment.

[10] How to interpret identical parameter names or message names in the scope of one SD is not defined or discussed in the MSC 2000 Standard [IT99].

[11] We could also define explicit broadcast capsules equipped with SPPs. However, the drawbacks arising from the hiding of communication structures would still remain.

Because of these methodological shortcomings and because of its dependency on UML-RT's current toolset, SAPs are an inappropriate solution for the modeling of broadcasting.

As we have outlined in Section 1, we aim at a clear notion of a component and its interfaces, as indispensable ingredients for defining and representing software architectures. Therefore, we stick with UML-RT's component, interface, and communication model, and integrate broadcasting into this model by means of explicit components within the software architecture we aim at. We still face two options on how to model broadcast mediums in this approach: First, we could model the medium by a container capsule which contains all capsules participating in the broadcast communication. As an example, consider Figure 15. This approach leads to a compact way of modeling broadcasting, and its hierarchical structure sustains clarity also for complex communication structures with several local broadcast mediums. The hierarchical structure facilitates dynamic communication structures (if desired) since the scope of capsules which possibly participate in a broadcast communication is controlled by the container which models the medium; thus new connections can easily be established. It also coincides with notions of refinement; refining a capsule into a structure which contains broadcasting just means to replace the component by a container for the broadcast medium and to delegate all tasks of the refined capsule to sub capsules. However, this approach may be inadequate if a hierarchical component should carry out additional functionality, or if the hierarchical structure established by the communication structure is not adequate from a functional point of view.

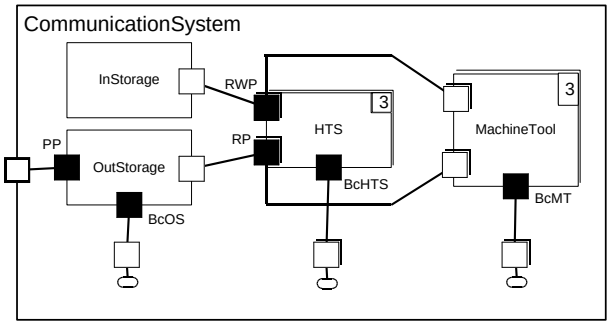


Figure 15: Modeling the broadcast medium as a container capsule

To gain more flexibility, we suggest a variant of the recursive control and subsystem controller patterns described in [Sel98] and [DW98], respectively. Consider Figure 16 for an overview of the basic structural decomposition we associate with components in our architecture. We distinguish three kinds of components:

- *leaf components*, which form the leafs of the component hierarchy,

- *broadcasting components*, which play the roles of containers for leaf components, nested broadcasting components, and an IO system,
- *IO system components*, which act as mediators for broadcasting messages received or generated by their container.

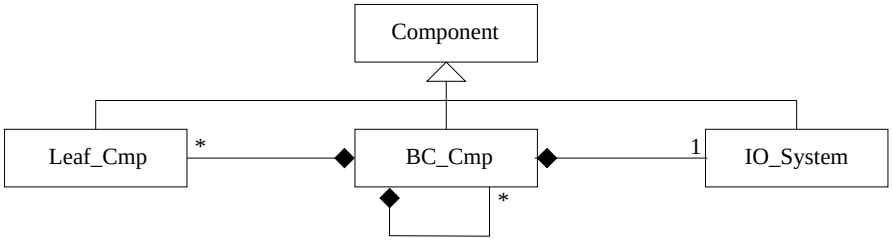
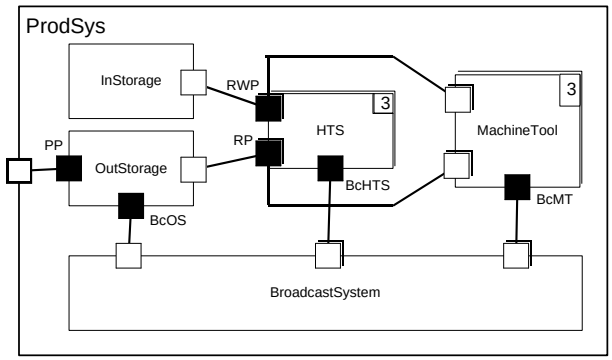


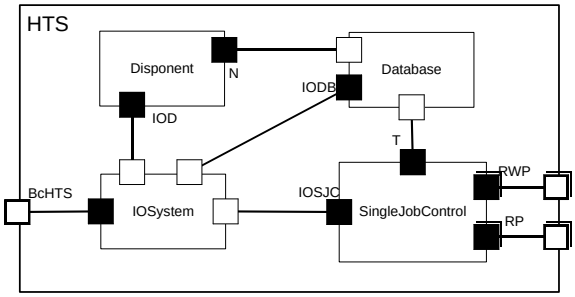
Figure 16: Architectural pattern for broadcasting

The idea is that every hierarchically decomposed component has an IO system which handles the broadcasting of messages among the relevant subcomponents of the container, and between the container and its environment. Intuitively, to perform a broadcast, a component sends a message to the IO system of its container. This IO system is then responsible for distributing this message to all other relevant components within the container, as well as to the IO system of the container's parent in the component hierarchy. This ensures that all broadcast messages reach all components participating in the broadcast communication. As an example, consider Figure 17. At top level view (a), we have the component `ProdSys`, which includes the components `HTS`, `InStorage`, `OutSorage`, `MachineTool` and a component `BC`, which is responsible for the delivery of broadcast messages. Within nested components such as the HTS, we have again a component IOSystem. We have no local broadcasting within the HTS; thus, the IOSystem only has to transfer broadcast messages from and to the subcomponents to the environment of the HTS.

We note several benefits of this architectural pattern. First, the hierarchic structuring of broadcasting components enables direct application of classical techniques for top-down structural system design. From the viewpoint of broadcasting, to refine a component structurally simply means adding a new IO system, and connecting it properly to the refining sub-components, and to the container's environment. This is directly supported by UML-RT's component and interface notion. Second, beyond their mere purpose of being mediators for broadcast messages the IO systems can also filter messages irrelevant for a particular subtree in the component hierarchy; this can lead to more efficient design and implementation strategies. Furthermore, components not participating in broadcasting need not have connections to IO system components at all. Third, the switch from broadcasting to other communication paradigms is immediately possible by a simple redefinition of the purpose of the IO systems.



(a) At top level



(b) Within nested components

Figure 17: Modeling the broadcast medium as an additional component

In the following paragraph, we show that this pattern seamlessly integrates with the extension of sequence diagrams we use in the requirements analysis, and how the resulting capsule structures can be derived automatically from requirements scenarios.

5.2 Systematic Derivation from Scenarios

In the following, we suggest a method for developing structure diagrams using the knowledge captured by the SDs specified during the requirements analysis. 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 serves 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 to get a first sketch of a structure diagram. 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. (a) Create a capsule for each class which appears in the SDs as an axis.
   (b) Create a capsule which performs the broadcast message passing, if broadcast communication occurs between axes in the SDs under consideration. We call this capsule *broadcast capsule* in the following.
2. Create a container capsule which contains the capsules from step 1[12].
3. (a) Create a binary protocol for each pair of capsules which exchange regular 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 broadcast capsule.

After these steps we obtain a first sketch of the structure diagram. Because we used scenarios to obtain the structure diagram, we could not expect that the diagram is complete and optimal. Furthermore the communication architecture may not be homogeneous. Hence in most cases, we have to modify the first sketch manually.

5.3 Deriving capsules for the transport system

In this section, we use the steps described above to develop the structure diagram of our holonic transport system. From the sequence diagrams we obtain the capsules `HTS`, `InStorage`, `OutStorage` and `MachineTool` (step 1a). The broadcast medium is modeled by the capsule `CommunicationSystem` (step 1b). These capsules are embedded in a container capsule `ProdSys` (step 2).

The capsule pairs HTS ↔ InStorage, HTS ↔ OutStorage and HTS ↔ MachineTool exchange handshake messages. Therefore the binary protocols RequestWP, RequestPlace and Request are created (step 3a). Table 1 shows the protocols from the view of the capsule HTS. The view of the corresponding communication partner is obtained by *conjugating* the protocols, i.e. interchanging sent and received messages.

[12]UML-RT requires a top-level container for all capsules of the system.

RequestWP	
send:	requestWP(wp)
receive:	releaseWP(wp)

RequestPlace	
send:	requestPlace
receive:	releasePlace

Request	
send:	requestWP(wp)
receive:	releaseWP(wp)
send:	requestPlace
receive:	releasePlace

Table 1: Handshake protocols

The generation of protocols is a bit more complicated: Since each of the participants in the communication receives all messages but may send only an individual subset of these messages, an individual protocol needs to be generated for each capsule involved in broadcast communication. Table 2 shows the broadcast protocols of HTS, MachineTool and OutStorage (step 3b).

To establish the actual component interfaces, each capsule gets assigned its respective ports associated with the protocol roles described above (step 4). For instance, HTS obtains four public ports which are associated to the base protocol roles RequestWP, RequestPlace, Request and BroadcastHTS.

Finally the connectors between the related handshake ports and between broadcast ports and container capsule are added (step 5). The result is the structure diagram depicted in Figure 18. Note that the five steps to generate the structure diagram could be performed automatically.

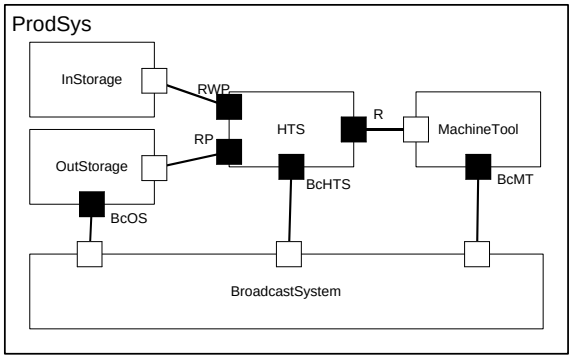


Figure 18: First sketch of capsule ProdSys

As mentioned before the obtained structure diagram may be not com-

BroadcastMT	
send:	jOrder(jobno, buffno, w, s)
send:	jEndOfNegotiation(jobno)
receive:	requestProdPrg
receive:	ProdPrg(p)
receive:	jOrder(jobno, buffno, w, s)
receive:	jBid(jobno, h, v)
receive:	jEndOfNegotiation(jobno)
receive:	jTransporting(jobno, h, w, wp)
receive:	jFinished(jobno, h, w, wp)

BroadcastHTS	
send:	requestProdPrg
send:	jBid(jobno, h, v)
send:	jTransporting(jobno, h, w, wp)
send:	jFinished(jobno, h, w, wp)
receive:	requestProdPrg
⋮	⋮
receive:	jFinished(jobno, h, w, wp)

BroadcastOutStorage	
send:	ProdPrg(p)
receive:	requestProdPrg
⋮	⋮
receive:	jFinished(jobno, h, w, wp)

Table 2: Broadcast protocols

plete or does not match completely with our imagination about the system. Not surprisingly, transforming the generated structure diagram into a complete and stable architecture description is subject to further subsequent steps.

The first step we have to carry out is to specify the cardinality of the capsules in the diagrams. According to the informal specification of the system, we specify the existence of three HTS and MachineTool capsules and appropriate replications of the ports of components communicating with these. Figure 19 shows the corrected diagram.

As second step, we perform a few optimizations. We replace the protocol `Request` and the respective ports by the use of the protocols `RequestWP` and `RequestPlace`, because the protocol Request is the only union of RequestWP and RequestPlace. Furthermore, we introduce the protocol

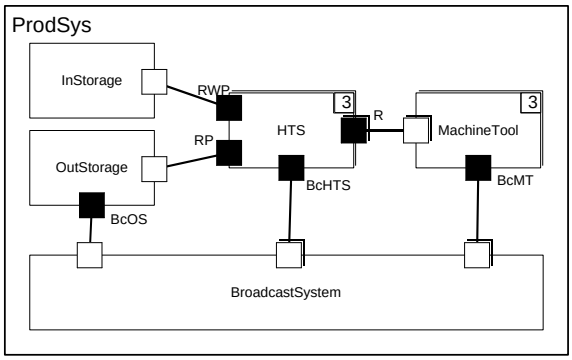


Figure 19: Capsule ProdSys with corrected cardinalities

`ProductionProgram` shown in Table 3 which allows us to set the ProductionProgram in the OutStorage.

ProductionProgram
send: setProdPrg(p)

Table 3: Protocol ProductionProgram

Finally, we obtain the completed and optimized structure diagram of CommunicationSystem shown in Figure 20. In the following sections, we refine the capsule `HTS` into appropriate sub structures.

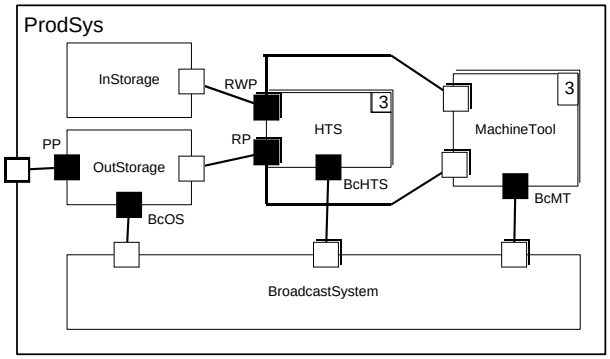


Figure 20: Final version of capsule ProdSys

6 First iteration: Refining the HTS

6.1 Objects in the Domain: A Class Diagram

In this paragraph we develop the substructure of a single HTS. An already known component of a HTS is the database. The database stores all the data which the HTS needs to perform its tasks. If we look at the informal description and the use case diagrams, we can derive two main tasks of an HTS: the negotiation of jobs and the execution of a job which includes the driving and the handshakes. Hence, we introduce a class `Disponent` which is responsible for the negotiation and a class `SingleJobControl` which is responsible for the execution of the current job. Following the design pattern for broadcasting introduced in Section 5.1, we introduce a fourth component called `IOSystem`. IOSystem handles the communication between the broadcast channel and the other components. It is responsible for the forwarding of broadcast messages to the CommunicationSystem and vice versa. Hence, IOSystem is associated to Disponent, Database and SingleJobControl, respectively.

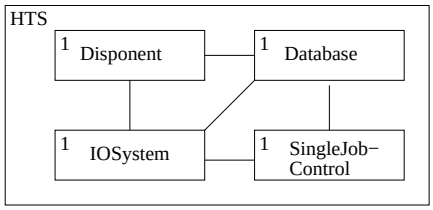


Figure 21: Domain of the HTS

Disponent and SingleJobControl get data from the database. Thus, they have an association to the database. The SingleJobControl performs the handshakes between the HTS and the MachineTool capsules, the InStorage and the OutStorage which is also expressed by associations. The obtained class diagram is shown in Figure 21[13].

6.2 Description of the Use Cases: sequence diagrams

In the following we develop sequence diagrams which show the communication between the components of one HTS. Each of the following sequence diagrams has to be consistent with its corresponding sequence diagram in the overall view (described in section 4.3.2), i.e. every message in a sequence diagram of the overall view must exist in the decomposed view as message to/from the environment and the order of the messages has to be

[13]For clarity, we used graphical containment to indicate the composition relationship between the HTS and its subcomponents. This variant is also supported by the UML and frequently used e.g. in [Dou98]

correct, too. In the decomposed view we use also the notation of broadcast SDs which we have introduced in section 4.3.2, but all broadcast messages end and originate at the lifeline of the IOSystem which manages the communication between the capsule CommunicationSystem and the HTS.

Let's look at the simplest sequence diagram `init`, focusing on the lifeline of the HTS with originator role. The HTS sends the message `requestProdPrg` and receives a message `ProdPrg(p)`. Thus, these messages must appear in the decomposed sequence diagram `HTSInit` depicted in Figure 22 as messages which are sent and received to the environment, respectively. Messages which are sent to or received from the environment are drawn with arrows or broadcast lines which end or originate at the bordering box. The resulting SD shows that the received production program is stored in the database.

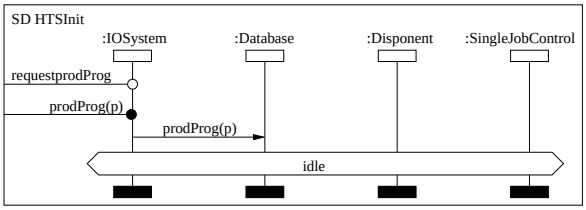


Figure 22: SD HTSInit corresponding to SD Init

The sequence diagrams in the Figures 23 and 24 refine the sequence diagram NegotiationOfOrder (Fig. 11). At the beginning, these sequence diagrams show the same scenario: after the message `jOrder` was received from the environment by the IOSystem, the latter forwards it to the disponent and the database. The disponent calls the database for the data needed for the computation of its bid. After receiving the data, it computes a bid v for the posted job, depicted by an action box. At this stage it is left open how the computation works. Note that during the computation other HTS may send arbitrary many bids for the job. Respective `jBid` messages are received by the IOSystem and forwarded to the database, expressed by the loop statement. The database stores the corresponding bid value v^* and the respective HTS h^* if the bid is better than the previous one. After the disponent has finished the computation, it asks the database: what is currently the best bid? Depending on the result of the query the disponent proceeds the negotiation or cancels it.

The scenario in Figure 23 depicts the case that a better bid of a other HTS is stored in the database, expressed by the condition $v' \leq v$. Hence the disponent does not send his bid and the idle condition is reached after it receives the message `jendOfNegotiation(jobno)`. We refer to this scenario as the listener role (thus the SD is named `HTSNegotiationOfOrderL`).

The SD shown in Figure 24 shows that the disponent has computed a better bid (condition $v' > v$). Hence, it sends its bid v to the IOSystem, which forwards it to the environment and to the database. After the re-

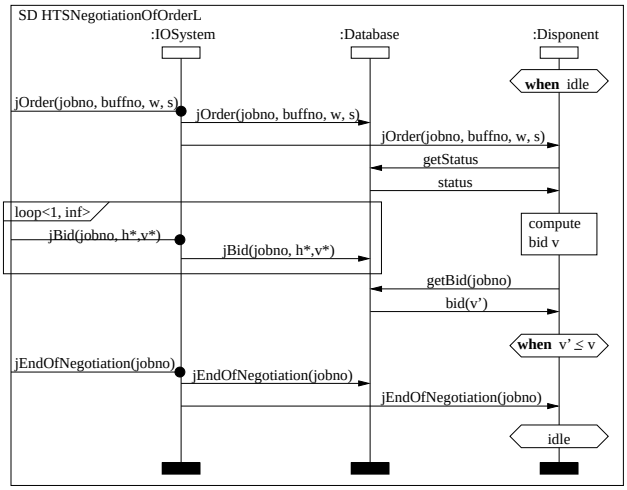


Figure 23: SD HTSNegotiationOfOrderL (listener role)

ceipt of the message `jendOfNegotiation(jobno)` the disponent asks the database which HTS has made the best bid because there may have been other bids in the meantime, depicted by the second loop box. If it is itself – expressed by the condition $h' = h$ – then it sends the message `inqueueJob` to the database. Otherwise only the idle condition is reached, without sending any further message. We refer to this scenario as the originator role.

Let us turn to the scenarios describing the execution of a job, i.e. transporting and handing over the work pieces. As mentioned before, the execution of jobs is carried out by the capsule SingleJobControl. Thus, most messages originate from and end at the respective lifeline.

In the SD shown in Figure 25, SingleJobControl requests the source and destination locations `location1` and `location2` for the next job to be executed from the database by sending the message `getNextJob`. Depending on the result of the query, the condition `idle` or `newJob` is reached.

The SD shown in Figure 26 describes the actual transport: If the condition `newJob` holds, SingleJobControl starts to drive to location1 (depicted by an action box). Then the workpiece is taken over, modeled by the messages `requestWP(wp)` and `release(wp)`, and the other HTS are informed of the start of the transport to the destination by a broadcast message. Here upon the condition transport is reached by SingleJobControl.

The SD in Fig. 27 analogously shows the completion of the transport: After driving to location2, the workpiece is handed over and the broadcast message `jFinished(jobno,h,w,wp)` is sent. Thereupon the database deletes the finished job. Finally SingleJobControl returns to the condition idle and waits for the next job (see 25).

For the sake of completeness we add the SDs shown in Fig. 28. They describe the receipt of the `jTransporting` and `jFinished` messages by the

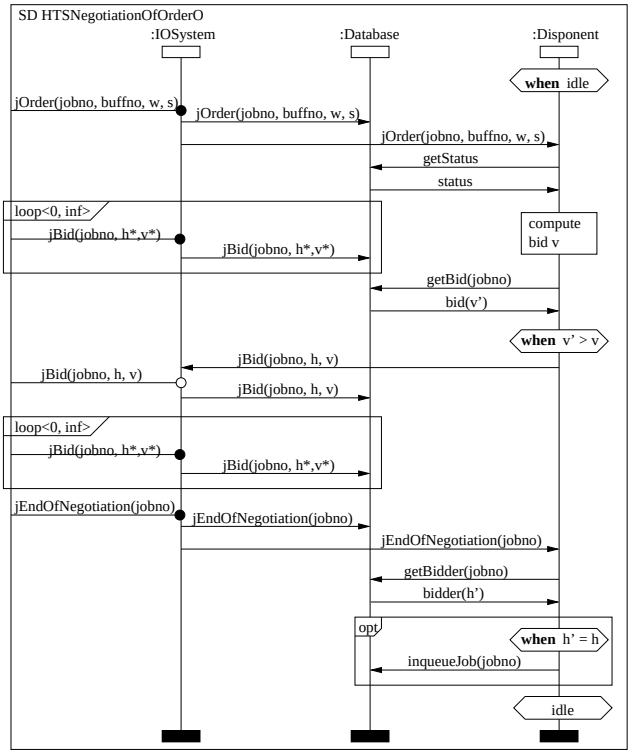


Figure 24: SD HTSNegotiationOfOrderO (Originator role)

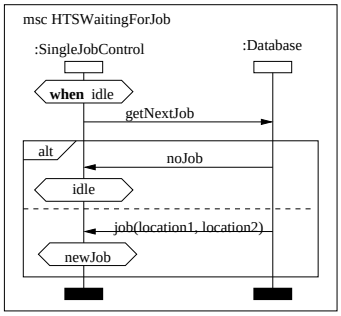


Figure 25: SD HTSWaitingForJob

other HTS.

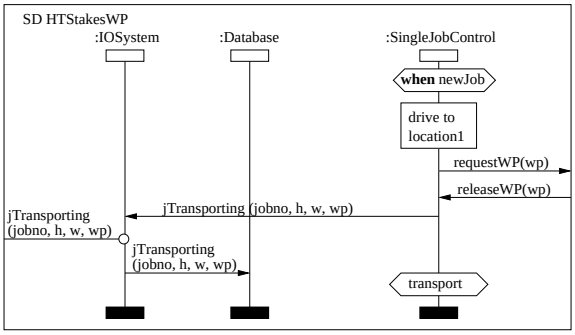


Figure 26: SD HTStakesWPintern corresponding to SD HTStakesWP

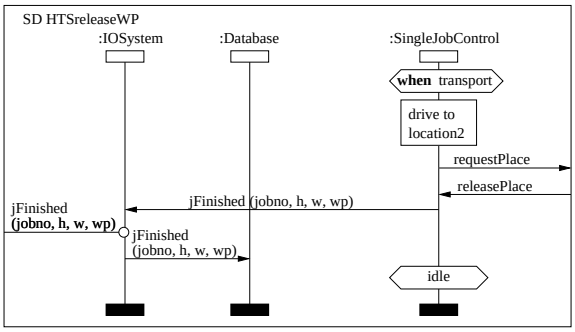


Figure 27: SD HTSreleaseWPintern corresponding to SD HTSreleaseWP

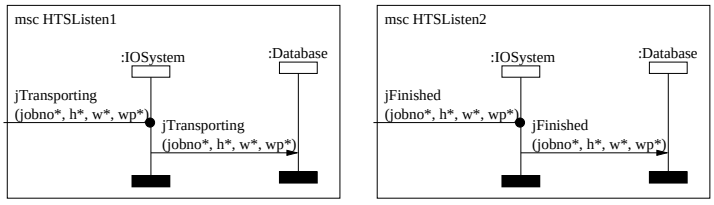


Figure 28: SD HTSListen1/2

6.3 Describing the HTS architecture: Capsules

In this section we sketch the derivation of the structure diagram for the HTS capsule from the sequence diagrams discussed in the previous section. We perform the same steps as in Section 5.2.

As before, we introduce a capsule for each class which appears as a lifeline in a sequence diagram of the HTS domain. These are IOSystem, Disponent, Database and SingleJobControl (step 1). The container capsule HTS with its public ports has already be derived in Section 5.2 (step 2).

IOSingleJobControl	
send:	jTransporting(jobno, h, w, wp)
send:	jFinished(jobno, h, w, wp)

IODisponent	
send:	jBid(jobno, h, v)
receive:	jOrder(jobno, buffno, w, s)
receive:	jendOfNegotiation(jobno)

IODatabase	
receive:	ProdPrg(p)
receive:	jOrder(jobno, buffno, w, s)
receive:	jBid(jobno, h, v)
receive:	jEndOfNegotiation(jobno)
receive:	jTransporting(jobno, h, w, wp)
receive:	jFinished(jobno, h,w, wp)

Negotiation	
send:	getStatus
send:	getBid(jobno)
send:	getBidder(jobno)
send:	inqueueJob(jobno)
receive:	status
receive:	bid(v)
receive:	bidder(h)

Transport	
send:	getNextJob
receive:	noJob
receive:	job(location1, location2)

Table 4: Protocols within the HTS

Next, we observe the communicating capsule pairs: IOSystem communicates with all other capsules within the HTS. Thus, we create the protocols IODisponent, IODatabase and IOSingleJobControl w.r.t. the communications with disponent, database and SingleJobControl. Additionally, we have to create protocols for the communications disponent $\leftrightarrow$ database and database $\leftrightarrow$ SingleJobControl (step 3a). All protocols are shown in Table 6.3.

At last, we need to consider the communication pairs IOSystem $\leftrightarrow$ environment and SingleJobControl $\leftrightarrow$ environment. These communications are delegated from the HTS to its sub capsules. Thus, the appropriate protocols BroadcastHTS, RequestWP and RequestPlace have already been derived in Section 5.2.

As in Section 5.2, we assign to each capsule ports associated to appropriate protocol roles (step 4). Further, we add connectors between the capsules IOSystem, Disponent, Database and SingleJobControl. Information on how to connect the ports `RWP`, `RP` and `BcHTS` is taken from the diagram derived in Section 5.2: The port `BcHTS` handles broadcast messages which are forwarded from and to the respective capsules by IOSystem, and `RP` and `BcHTS` are connected to the public ports of the HTS handling the takeover of workpieces (step 5).

From these steps, we obtain the structure diagram shown in Figure 29)[14].

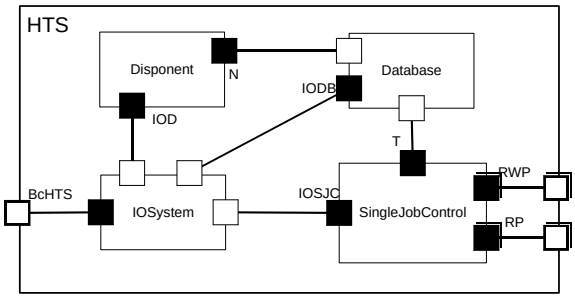


Figure 29: Structure diagram of capsule HTS

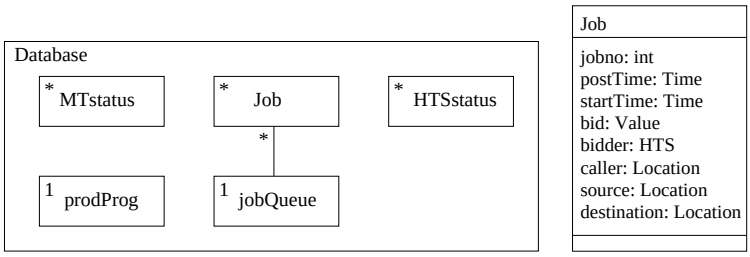
[14]For brevity, the protocol names are omitted in Figure 29.

7 Second Iteration: Refining the Capsule Database

In this section, we turn our attention on the capsule `Database`. Clearly, for a detailed design we are interested in the data structures which represent the concepts of jobs, machine status information and so on. Especially, we are interested in the distribution of the data, i.e. which information is kept locally by each HTS database. This is a crucial matter since we need to know, e.g. for later optimizations - which information can be received from the database and whether it is reliable or not.

At a first glimpse, we might expect that the data modeling is the only interesting part of the database model, for the database just stores information and delivers it when demanded later on. However, as we will see both in section 7.3 and section 8, there are also interesting behavioral aspects associated with the data model. In the following, we will first refine the data model and the use cases in which the database is involved. We will also consider the component structure of the database capsule.

7.1 Information Model of the Database: A Class Diagram



(a) Database (b) Job

Figure 30: Domain of the database

The class diagram in (30a) shows the conceptual data structures which are contained by the database class. Clearly the database has to store an arbitrary number of `Jobs` being currently negotiated or executed. For the jobs which are to be executed by the "own" HTS (accepted bids), it is convenient to have a separate data structure - one `jobQueue`[15]. For the computation of bids, the database stores status information about the HTS

[15] A star in the left upper corner of the Job class indicates that there are zero or arbitrary many jobs stored in the database. If the quantity is known, it is specified by a number specifies as for the jobQueue.

itself and about the machine tools. The latter enables the HTS to determine whether a job can be carried out without delay or if it has to wait the second machine/ storage unit involved in the job. Finally, the database stores the informations about the production program `prodProg`. As mentioned above, this class diagram shows *conceptual* data structures which need to be refined in the course of later development steps. For the moment however, it fits our needs.
Figure (30b) shows the attributes of the class Job. Of course it contains the relevant information about a job, such as job number, points of time, bid etc. Let's have a closer look at the attributes caller, source and destination. The Attribute `caller` is a reference to the location which sent the jOrder-message. Source and destination are the locations between the workpiece has to be transported. Hence we have to express that caller is a machine tool and that the same machine tool has to be either the source or destination.

Constraints The constraint is expressed in OCL:

```
context Job inv:
caller.oclIsTypeOf(MachineTool) and
(caller = source xor caller = destination)
```

7.2 Component Structure of the Database: A Capsule Diagram

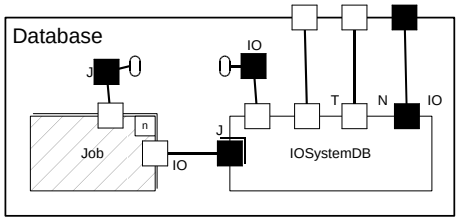


Figure 31: Capsule diagram of the database

In some cases, before considering the scenarios taking place within a refined component, it is helpful to develop a complete specification of its internal structure. The database keeps three communication relations and contains a conglomerate of jobs, job related data and status information. For the interests of neighbor components in the information stored in the database partially overlaps, a clear communication concept is necessary. Thus, we first develop the internal structure of the database and refine

the SDs in which the database is involved afterwards, w.r.t. this structure. From these, protocols can be derived as shown in Section 5.2 (since the generation trivially follows the scheme presented in Section 5.2, it is omitted here).

The capsule diagram in Figure 31 shows that we model each job stored in the database as a capsule. All other information is maintained as data objects directly by the database. Since the database receives broadcast messages via the HTS, we apply the pattern presented in Section 5.1 and add an IOSystem to the subcapsules of the database. The existence of an IOSystem within the database may seem superfluous at a first glimpse but has several advantages: First, it facilitates further refinements of the database, e.g. by defining additional subcomponents. Further, it divides the concerns of the database itself and the contained jobs. Whereas messages affecting the data of a particular job are handled by the respective job capsule, messages concerning the management of jobs - creation and destruction - and of status information are processed by the database itself. In order to separate the concerns of the database and its jobs consequently, we apply a slight extension of the IOSystem pattern: Within the database, the IOSystem additionally handles the messages sent and received along the port `T` and `N`. This structure enables the derivation of comprehensible and compact behavior models which are discussed in Section 8.

Note that drawing this capsule diagram is not a fundamental change of the development steps described in the previous sections. We do only identify capsules and bindings here, which is also possibly a *partial* view on the database. It structures helps structuring the refinement of the interaction scenarios in which the database is involved, but can also be refined itself during these development steps. In particular, no protocol is defined in advance. Protocols have to be derived from the sequence diagrams, using the same generation steps as described above.

7.3 Refining the Use Cases of the Databases: Sequence Diagrams

In the following, we refine the use cases scenarios in which the database is involved. We will only consider scenarios here which require nontrivial internal interaction within the database, that is, interactions which affect jobs. All remaining scenarios are assumed to treat incoming messages by just storing their values appropriately and outgoing messages by sending the according attributes from the database.

Figure 32 reveals most of the interaction with the environment is delegated to capsules of class `Job`. Thus, 32 refines the sequence diagram shown in 24 in a straightforward way. The refinement of the sequence diagram in Figure 23 is quite similar and omitted here.

The database is responsible for the creation and destruction of jobs, and for the maintenance of status information which does not belong to a

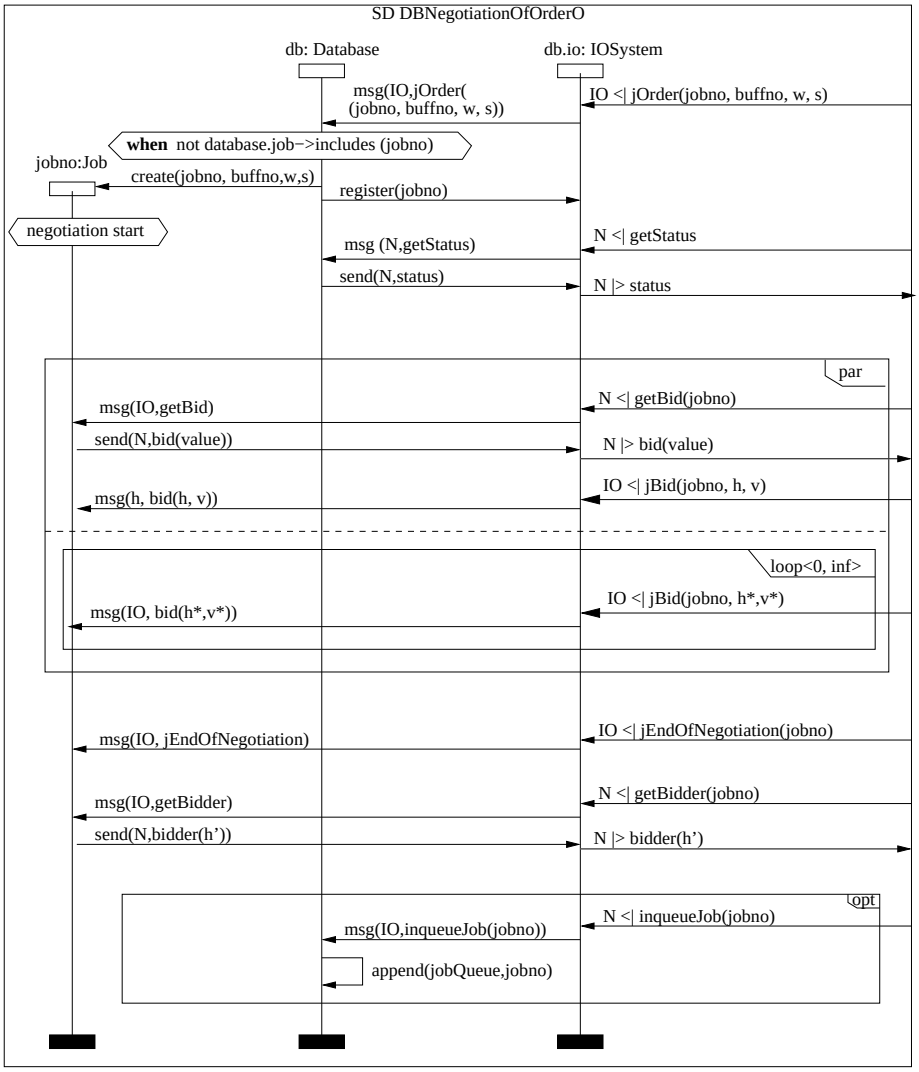


Figure 32: Negotiation from the databases viewpoint

particular job[16]. The forwarding of messages from and to jobs is carried out by the IOSystem. A message, together with an indication of the original sender, is forwarded to the database or the respective job if this job exists. Typically, the condition that this job exists is checked locally for each communication step by an implementation of the IOSystem. We could have refined this scenario by adding the appropriate conditions; however this would lead to unreadable SDs. In Figure 32 this constraint is superfluous because the job is created by the database before messages for the job are received. Yet, it is of interest how the whole database component behaves on the receipt of unexpected messages due to communication failures. This

[16]The maintenance of status information could also be delegated to dedicated capsules which would be more consequent. For brevity, we omit this decomposition.

part of the database behavior and appropriate conditions for the message forwarding by the IOSystem are specified in the scenarios shown below.

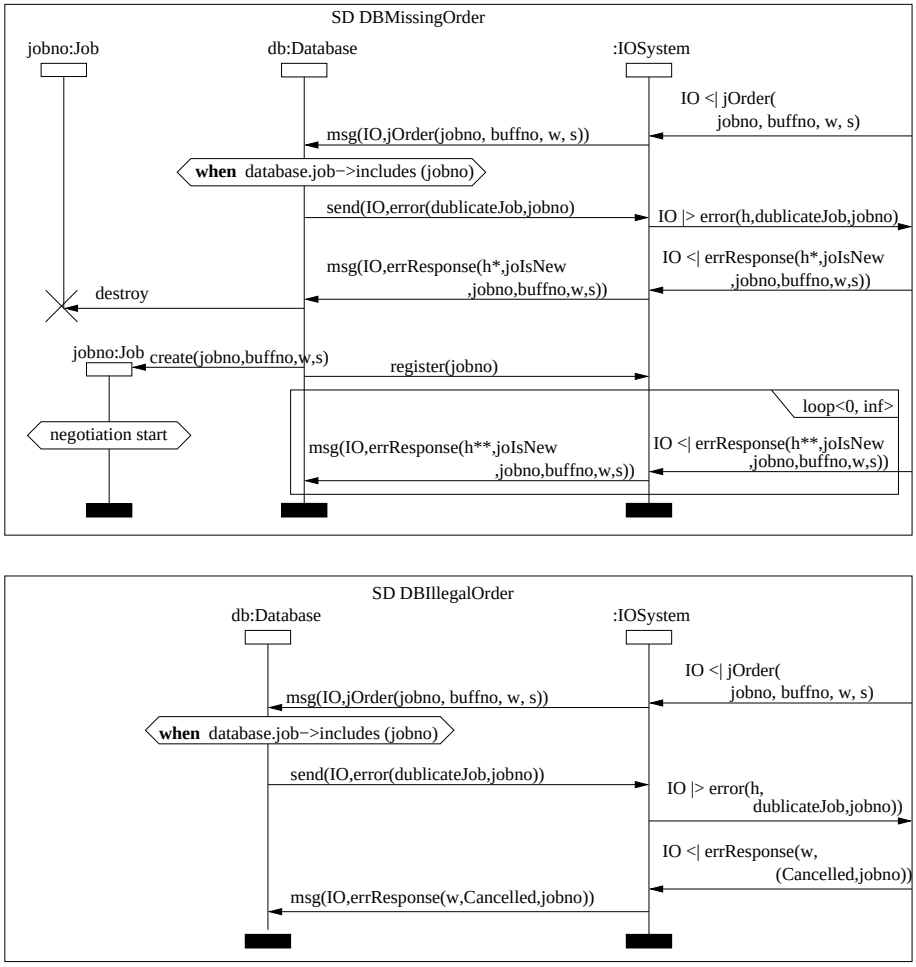


Figure 33: Duplicate jobs when an order arrives

Figure 33 considers situations when illegal bids are broadcasted, i.e. bids for jobs which are not known by all HTS databases. This may happen for the communication was disturbed earlier, an HTS was off-line or one of the components encountered an error. If the database encounters such an error by the receipt of a `jOrder` message[17], it requests the IOSystem to send broadcast message describing the error. If the conflict is due to the data in the database is not up to date, it gets corresponding messages by the other bases, revises it's information base, and creates the new job. Afterwards, we can proceed with the "normal" processing of the order[18]. If the database is in the role of receiving an error message which is confirmed

[17]The message `jOrder` triggers the creation of a job and is therefore handled by the database.

[18]We concentrated on the error handling here to keep the sequence diagrams short. The interrelation with the sequence diagram 32 will be considered in Section 8.

by an other HTS, the database recognizes that the job is no longer valid, destroys it and informs the disponent about the error.

A similar situation is addressed in Figure 34. Here, the database receives a status information or information request concerning a job which is not available in the base. Since the IOSystem which receives the job has no information how to forward the message, it notifies the database capsule about the missing job which responds with the request to broadcast an error message. Three scenarios are of interest:

The job may be valid, e.g. it has not been stored in the database regarded in the scenarios because of communication failures but is known by the databases of all other HTS. Some other component notifies the database of the existence of the job, providing the status information. The job is created and the processing of the order can proceed in the normal manner.

In the second case (SD `DBIllegalBid`), analogously, an error message for a bid which is considered to be illegal is generated. This time, the message is responded by the cancelation of the originator of the erroneous bid.

In the third SD, the bid was invalid. The bid has been generated by the disponent of the modeled databases container (HTS_i) and is considered to be valid by the database although the corresponding job does no longer exist in the databases of the other HTS. Since the bid is also received by the other HTS, the database receives error messages from them. By means of these messages, the database recognizes the error, deletes the erroneous job and informs the disponent about the error.

The refinement of the remaining scenarios in which the database is involved is straightforward. Figure 35 shows the interactions of the database during the execution of a job. The database capsule maintains the priority list of successfully negotiated jobs and is responsible for the destruction of a job capsule after the corresponding job has been finished. Status messages are forwarded directly to the corresponding job capsule. Of course, also the scenarios 25, 26, 27, and 28 have to be refined. Because this requires only trivial message forwarding or actions on local data structures which are not expressed by sequence diagrams, we omit an them here.

Constraints The sequence diagrams for the database pose problems w.r.t. constraining the values of component names and message parameters which are similar to the constraints discussed in Section 4.3.2. Obviously, we expect the database to send messages to the right job; similarly, the database should only create a job if a job with a given number does not exist. To specify these constraints, we access the subcomponents of the database by constraints like **when** `database.job->includes (jobno)`. This works if the `create` operation yields a component with an appropriate jobno. This is not defined by UML-RT's semantic so far, since the semantic of capsules doesn't refer to OCL, but this could be easily formalized.

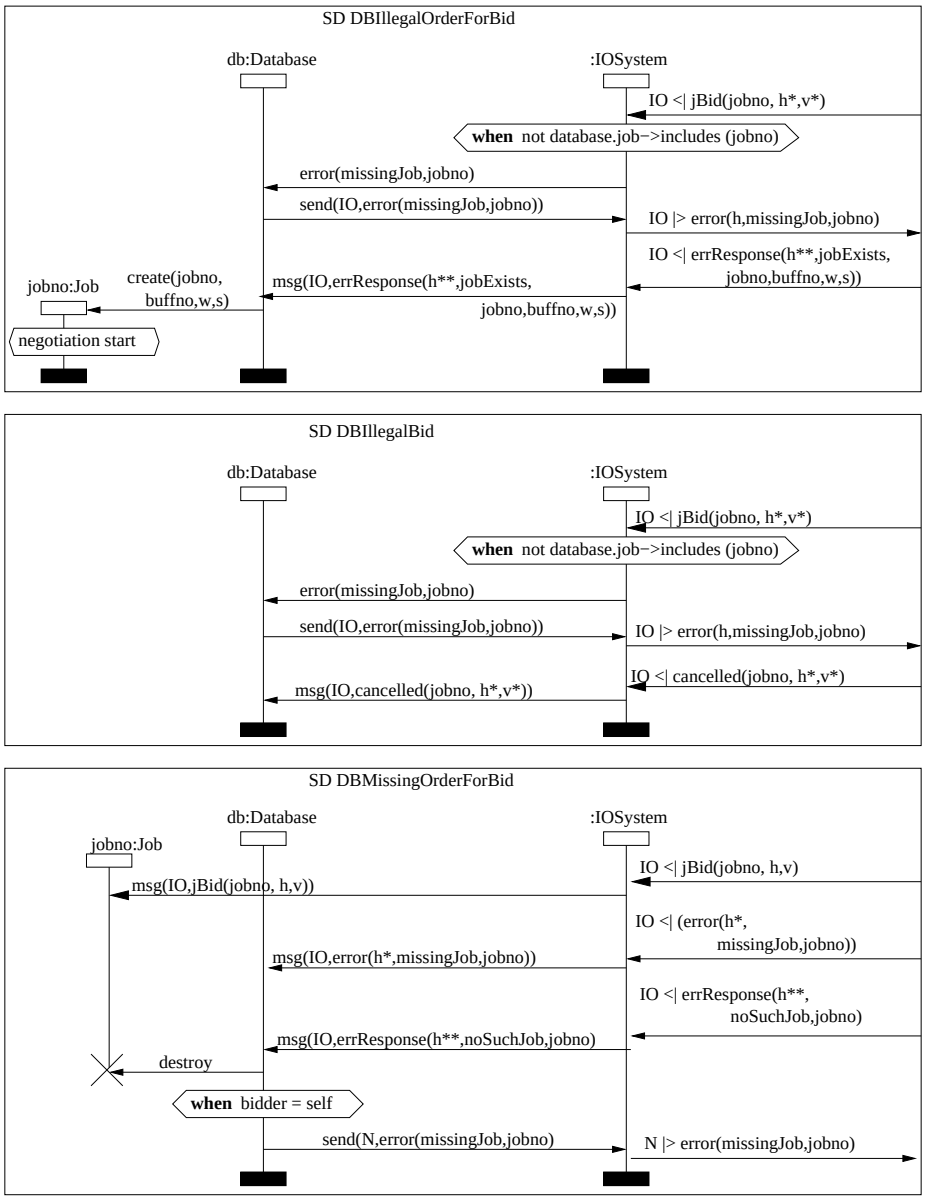


Figure 34: Missing jobs for bids

However, there is also the need for the specification of constraints which cannot be specified by condition labels, since they also affect data not visible in the diagram. For example, the message `getExecInfo` in the scenario `DBtakesWP` is only sent to the first job in the job queue. This could be specified by a condition like `database.jobQueue->first = jobno`. These constraints can be formulated by an extension of OCL's navigation mechanism to relationships between capsules.

Again, a problem is the scope of such constraints. Parameters like `jobno` must be defined to be variables for a particular sequence diagram.

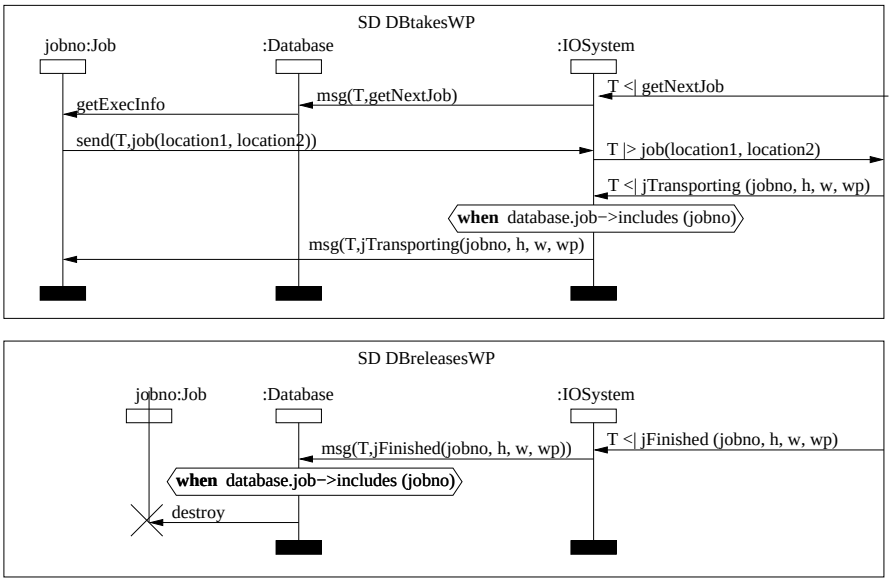


Figure 35: Response of the database on transport messages

Further, it has to be defined how such scopes of parameters are affected by the combination of two SDs via condition labels. Also, to support constraints like the one mentioned above, it is necessary to extend the notation of condition labels we have adapted from [IT96].

8 Third Iteration: Deriving Behavior Descriptions

In this Section, we develop behavior specifications for the capsules identified above, using UML-RT statecharts. Using the generation algorithm presented in [KGSB99, Krü00a], this development step seamlessly integrates with the development of scenarios and capsule structures presented in the previous Sections. Behavior models of the capsules can be generated fully automatically. The generated statecharts can mark the starting point for the development of the behavior of the capsules. The early models will usually be neither complete nor as detailed as needed for implementation purposes. Nevertheless, they cover all important aspects of the system to be developed. Using refinement steps for each aspect which has to be added or reconsidered, the generated models are transformed into more detailed and implementation specific ones. This is the actual work which has to be carried out by the developer.

In the following, we will focus on a high level design of the transport system. In particular, we will consider the automatic generation of statecharts from the sequence diagrams developed in the previous Sections. We will consider the conditions necessary for the generation, but also some intermediate results of the generation in order to discuss the adequateness of the algorithm for the UML-RT modeling languages. Further, we will discuss refinements which address constitutional design decisions to provide evidence that the generated models and UML-RT statecharts used in our approach are a useful combination within practical system development.

8.1 The Transformation Algorithm - an Overview

Before we turn toward the concrete models, we recall the basic steps of the transformation algorithm from and discuss some principal questions concerning its application in our setting. Details can be found in [KGSB99, Krü00a]. The generation algorithm consists of the following five steps:

Projection: We are to generate one statechart for each capsule. For behavior of the capsule, only the lifeline in the sequence diagrams which represent the capsule to be processed is of interest.

Normalization: The principal question in the generation is which interaction traces are defined by a given set of sequence diagrams, i.e. at which points in the execution traces the modeled scenarios occur. This information is taken from *conditions* which appear in the sequence diagrams: the corresponding scenarios can occur at a point in an execution trace if the state condition are fulfilled at this point.

To allow the sequence diagrams to be processed in a uniform way, they are normalized: Each SD has an initial and a postcondition. The

precondition must be met before the scenario starts, and the postcondition is established at the end of the SD. If a sequence diagram does not have a condition at the beginning or the end, a condition `initial` is added as prerequisite / postcondition. A SD starting with this condition can occur at the beginning of an execution, and at any point when it is established by the postcondition of the preceding scenario. If a sequence diagram has more than two conditions, it is split into two or more sequence diagrams with exactly two conditions.

Transformation into an SD-Automaton: Since conditions represent states and we have sequence diagrams which start and end in such a state, sequence diagrams can be taken as transitions to generate an automaton describing the high-level behavior of the capsule: the SD-automaton.

Transformation into an automaton: Each sequence diagram is replaced now by the concrete interactions. For each sent or received message, a single transition is generated. Each two subsequent transitions are separated by an intermediate state.

Optimization: The resulting automaton usually can be optimized in several ways. For example, standard algorithms like removing ϵ-transitions and/or making the automaton deterministic can be applied. Other optimizations include e.g. the merge of a set of transitions into a single transition. Examples will be discussed in the following sections.

8.1.1 Application of the Algorithm on UML-RT

The steps 1-4 of the algorithm can be applied almost straightforward to UML-RT sequence diagrams to generate UML-RT statecharts. Details regarding these steps are discussed in the following sections. Only the optimization step needs to be considered in-depth. Since it depends on the semantics of sequence diagrams and statecharts which optimizations are "legal". In this subsection, we discuss some optimizations which are legalized by the behavioral semantics of UML-RT. These optimizations are applied in the Sections 8.2 through 8.4.

UML-RT statecharts are based on an *asynchronous execution model*. This means, that the time instance when a transition fires is chosen in a nondeterministic manner. This is described in more detail in [San00][19]. The semantics ensures only that the transition fires *not before* the triggering conditions become true, and the delay is finite. Based on this semantics, we are allowed to merge subsequent transitions into one for additional conditions:

[19]Other variants of automata fix the time instance of the firing of transitions deterministic. For example, [Inc02, Gmb02] enforce specification of a tact rate of a discrete global clock. Each automata in a system fires on transition when the clock ticks. We use such an execution model in *InTime* when we move to implementation and verification in the development process.

- An action transition (e.g. a send operation) can be merged with an immediately preceding triggering transition (e.g. `rec Database bidder(h')`, if the intermediate state possesses no other incoming or outgoing transitions..
- Also –on the same condition– subsequent action transitions can be merged into a single transition provides that it preserves the ordering of the actions[20].
- If an intermediate state has two incoming or two outgoing transitions, as illustrated in Figure 36, then the transformations mentioned above are legal if they are legally carried out on both branches.

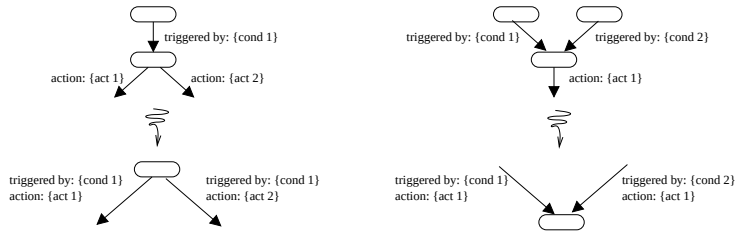


Figure 36: Examples for optimizations

- Of course, similar transformations are possible for more complex situations. We do not consider them here because they lead to significant structural changes of the resulting statechart. Thus, they can represent serious obstacles for traceability. If convenient, they can be applied manually.

The rules above eliminate superfluous states and transitions in the generated statechart. The resulting statechart is not minimal. We do not apply the known minimization algorithms and further optimization algorithms from automata theory like removal of ϵ-Transitions or non determinism (as shown in [HU79] and others) for methodological reasons:

We plan to extend our approach towards an incremental development process. The resulting statechart is only the starting point of the design process of the system's components. Thus, it will be refined in many further design steps. To be able to deal with common situations such as the addition or change of requirements during the design phase, we aim at structure preserving transformations that facilitate traceability of changes in scenarios and/or design.

For this reason, we impose a constraint on the application of the minimization steps presented above: No state generated from a condition label may be deleted by a minimization step.

[20]This optimization corresponds with common notions of refinement. This behavior corresponds to firing of the first transition with the same delay and the firing of the subsequent merged transitions without delay. This is a legal behavior for the non optimized statechart. Thus, the optimization leads to a more deterministic timed behavior.

8.2 Developing a Behavior Description for the Disponent

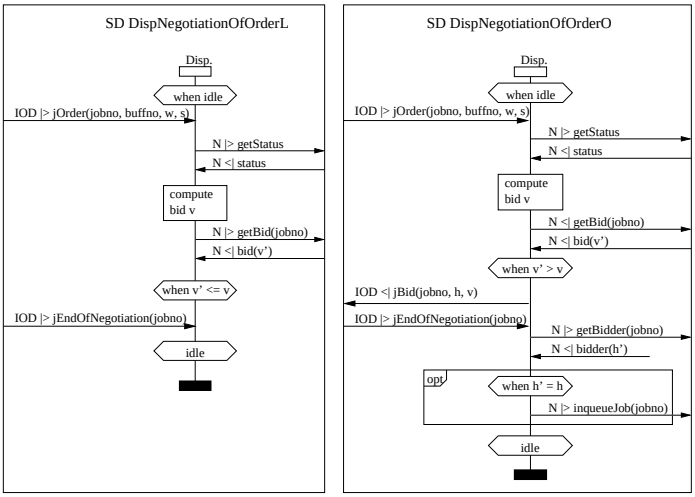


Figure 37: Projection of the scenarios on the lifeline of the disponent

For the generation of a statechart which specifies the behavior of the disponent, we have to consider only the sequence diagrams shown in Figures 23 and 24. In the first step of the generation, the sequence diagrams are projected on the lifeline representing the disponent, as illustrated in Figure 37.

The next step is to normalize the sequence diagrams. Start and end conditions are given. We only have to split the sequence diagrams into smaller ones because of the conditions $v' \leq v$, $v' > v$ and $h' = h$. Because the sequence diagram `HTSNegotiationOfOrderO` and `HTSNegotiationOfOrderL` coincide in their first part, we get one sequence diagram twice and can omit one copy. The treatment of logical conditions requires an extension of the generation algorithm in as presented in [KGSB99, Krü00a] where only condition names are considered. For condition labels which only contain logical conditions, we need to introduce condition names in order to maintain the ordering relation of the SDs shown in Figure 37. Unique condition names for each label could be generated automatically . However, since these conditions carry important state information, we suggest to perform this step manually by specifying suggestive names. A further minor extension is needed to treat alt- and opt-boxes: They need to be syntactically transformed into a set of sequence diagrams which represent the cases. This split is straight forward, thus we do not discuss this step in detail here. The result of the normalization is shown in Figure 38.

From these SDs, a SD automaton can be generated. Again, logical conditions require a slight extension of the generation algorithms. They always occur at the start of an sequence diagram as preconditions of the following

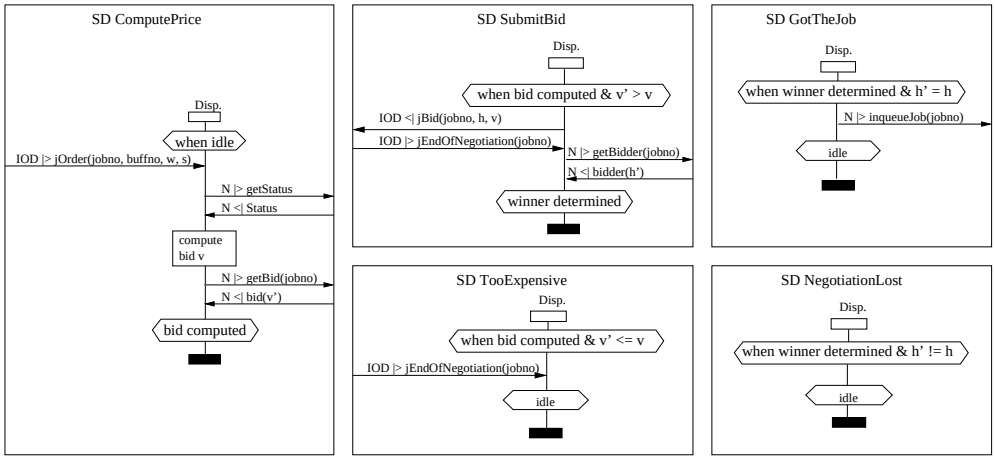


Figure 38: Adding missing state conditions to the sequence diagrams

sequence. Thus, they are attached as preconditions to the corresponding transition in the SD automation. For the disponent, this generation step leads lead to the SD automaton shown in Figure 39:

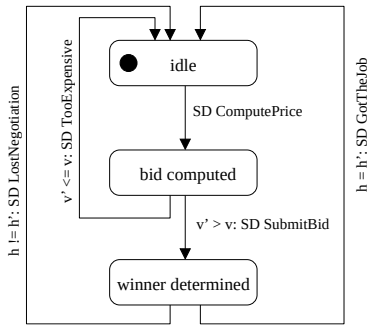


Figure 39: SD automaton for the disponent

This automaton is the basis of the generated statechart. The next automatic step is simple: Each transition labeled by an sequence diagram is replaced by a sequence of transitions and states. For transitions which also contain a precondition resulting from the extension of the generation algorithm, this precondition is added as a precondition to the first transition of the sequence by which the "sequence diagram transition" is replaced. This leads to the statechart shown in Figure 40.

The statechart generated by this simple algorithm already serves as a good starting point for the development of a statechart for the disponent. Still, some parts of the chart are unnecessarily complex. However, we can reach a significant reduction of the complexity by applying the optimizations discussed in section 8.1.1. Figure 41 shows the application of these optimizations on the generated statechart.

We omitted the optimization for one state: We haven't reduced the

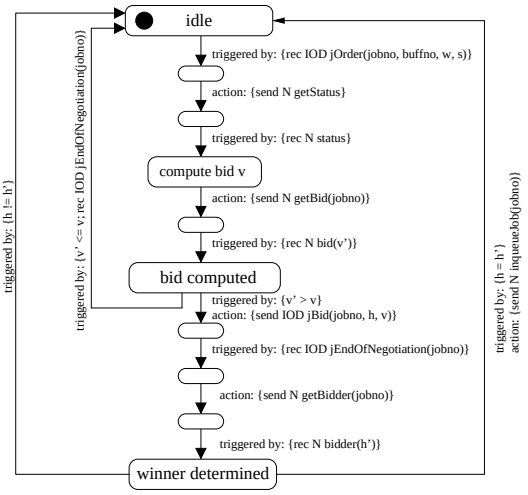


Figure 40: Generated statechart for the disponent

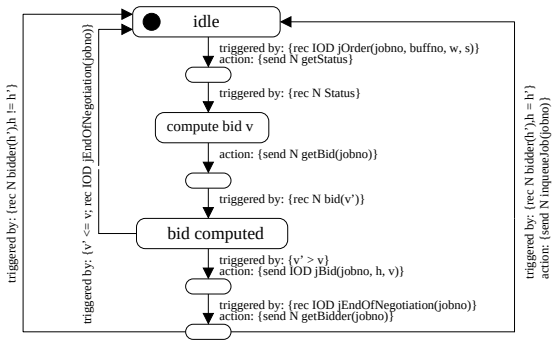


Figure 41: An optimized statechart for the disponent

state `compute bid v` because this state is subject to refinement in later development steps. Since such decisions need interaction with the developer, we suggest to perform the first four steps of the generation algorithm fully automatic, and perform the optimization step with tool support interactively.

8.3 A Statechart for the SingleJobControl

The generation of statechart for the capsule `SingleJobControl` starts with projections of sequence diagrams from the Figures 25, 26 and 27. The projected SDs are shown in Figure 42. The normalization of the SD `SJCWaitingForJob` and the following steps are straightforward, the generation yields the statechart shown in Figure 43.

Of course, this statechart can be refined in numerous ways. For example, we use a very simple communication protocol between the SingleJobControl and the database here. Surely, it is not appropriate for an

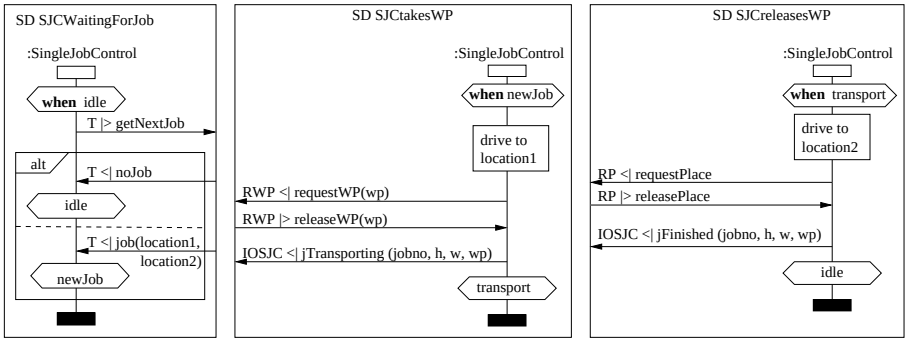


Figure 42: Input SDs for the generation of the SingleJobControl

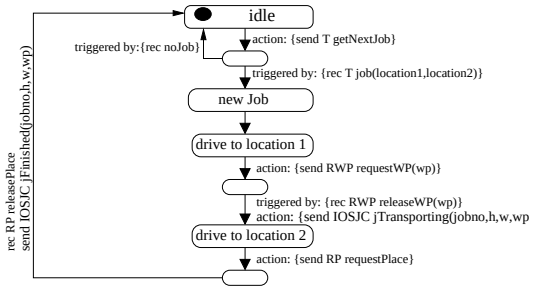


Figure 43: A statechart for the SingleJobControl

implementation since it involves busy waiting. However, for the moment we are interested in a conceptual model. Thus, the simplicity of the protocol gained by avoiding message queuing and timing concerns is convenient at this state of modeling. We can refine this protocol to an efficient one by applying interface refinement on the interface towards the database, letting the database actively signal when a new job has been successfully negotiated. We also will refine the states `drive to location1` and `drive to location2` into a sub chart which controls the traveling. This is subject to future work on the application of refinement calculi on UML-RT models, especially with respect to the treatment of real-time aspects.

8.4 A Statechart for the Database

In Section 7, we have refined the database into a container capsule which delegates a part of the interaction at its ports to a set of sub capsules: Job capsules maintain job information and the forwarding of received and sent messages is performed by the sub capsule `IOSystemDB`. The container capsule `Database` is responsible for the creation and deletion of job capsules and for the maintenance of job independent status information. Thus, the generation of behavior models needs to consider the respective capsules. In this presentation, we restrict ourselves on the generation of the behavior of the database and the job capsule. Since the database is involved in all

scenarios of negotiating and executing transport tasks, we have to deal with a more complex set of scenarios here.

8.4.1 The Database

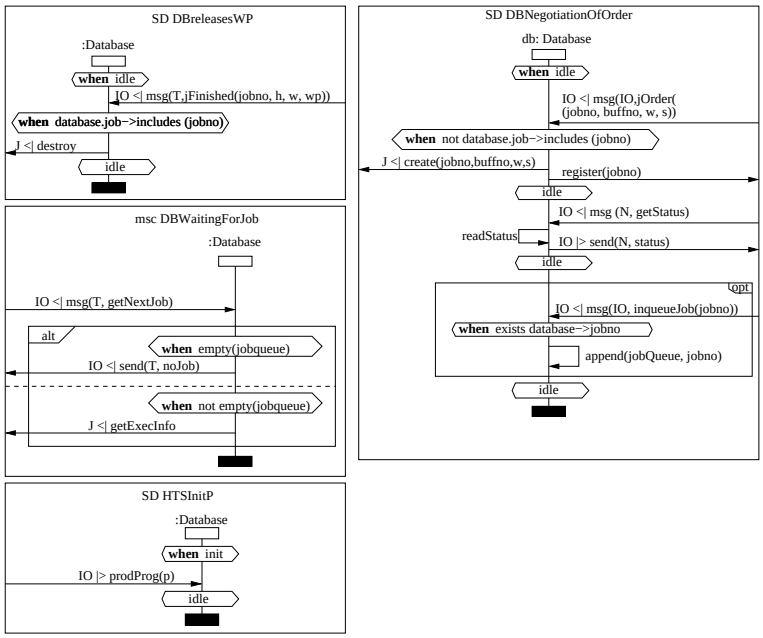


Figure 44: Refined SDs for the database

Let us start with the database: For this component, we have specified almost no state information until now, only that the sequence diagram `HTSInitialization` ends in the state `idle`. Leaving it with that for the generation would yield a statechart which was surely not intended: All sequence diagrams start in an initial state and end in the same state except `HTSInitialization` which would end the execution. At this point, the developer has to decide which ordering of the sequence diagrams is important for the database.

Similar to the components generated in the previous sections, we need to specify state information regarding the execution ordering of the SDs. However, the ordering of the actions of the database need not to follow the complete negotiation scenarios. For the database, it suffices to identify transactions which need to be carried out atomically to ensure consistency. This can be specified by adding explicit state conditions that the database returns to a listen state – also named `idle` – after each message request. Thus, we reconsider the sequence diagrams for the database, and specify the state conditions shown in the projection of SDs in the Figures 44 and 45. For brevity, we skipped the projection of the SD `HTSNegotiationOfOrderL` (Fig. 23) and of trivial sequence diagrams such

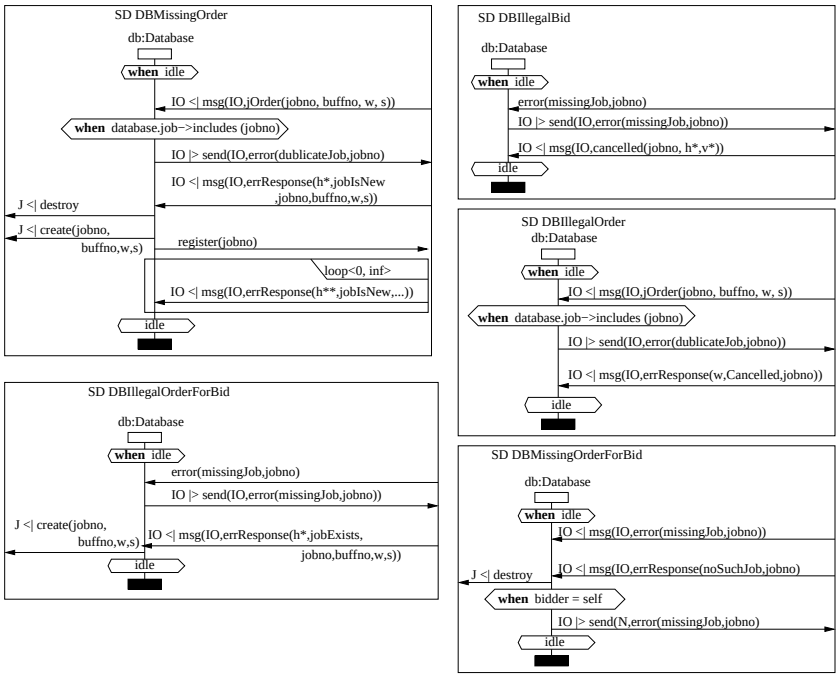


Figure 45: Refined SDs for the database

as `HTSListen1` in the presentation[21]. Since the order of the messages occurring in negotiations and transport tasks are not relevant for the database, they do not affect the state of the database. This is made explicit by adding condition labels that show that the database always returns to the idle state after the processing of each message.

From these scenarios, the steps 1 to 4 of the generation algorithm (without the optimization) yield the statechart shown in Figure 46. Obviously, this statechart is too complex for a readable specification. This reflects the fact that the database has to carry out by for most of the interactions of the components we have considered.

Fortunately, the complexity of Figure 46 can be reduced significantly: First, in many cases subsequent transitions can be combined to single transitions by the optimizations presented in Section 8.1.1, Figure 36. We defer this optimization step in support of a more powerful improvement: refining the statechart into a hierarchical statechart by wrapping single states and transitions into a hierarchical state. Carrying out this refinement before the optimization avoids that this step combines transitions which should belong to different levels of abstraction in the hierarchical chart. The optimizations are applied on the toplevel statechart and each sub statechart afterwards.

The Figure 47(a) shows the toplevel statechart of the refined behav-

[21]Please recall from Section 7.3 that the database capsule is not involved in the forwarding of job information. This is handled directly by the IOSystem and the respective job.

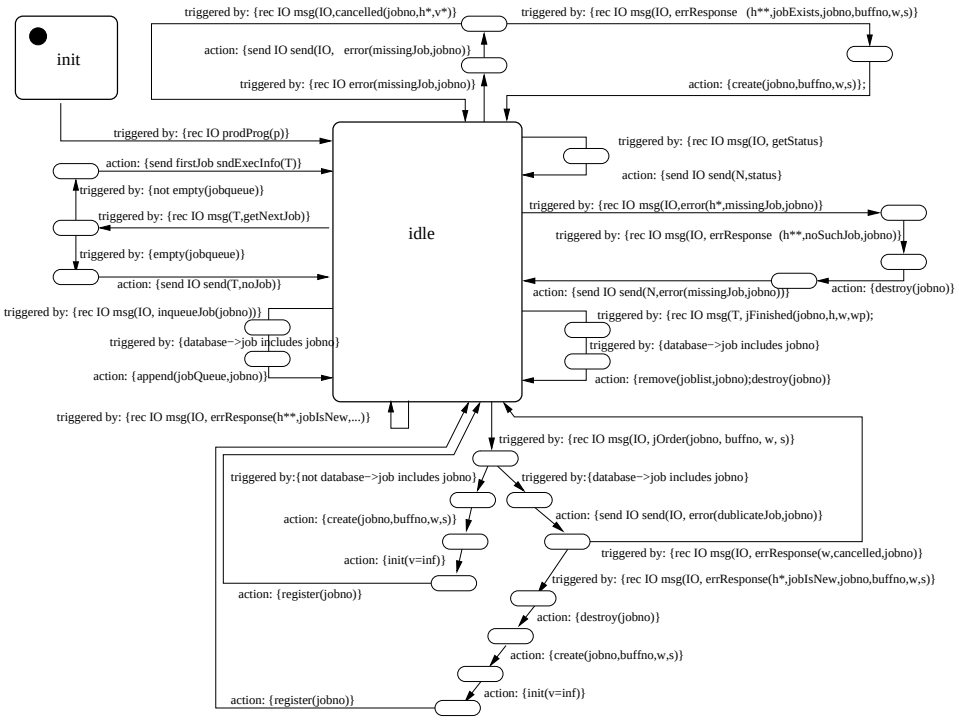
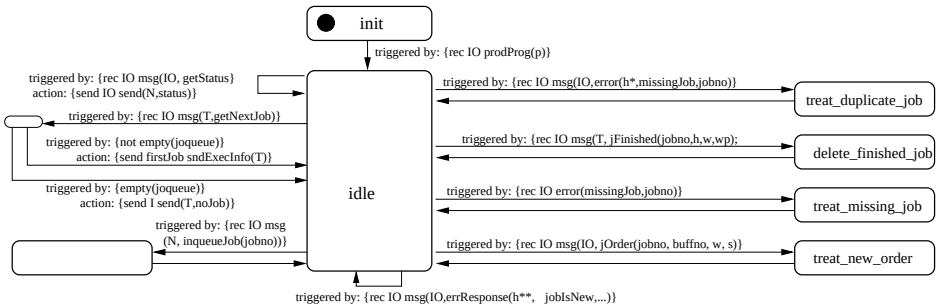


Figure 46: Statechart for the database without optimizations

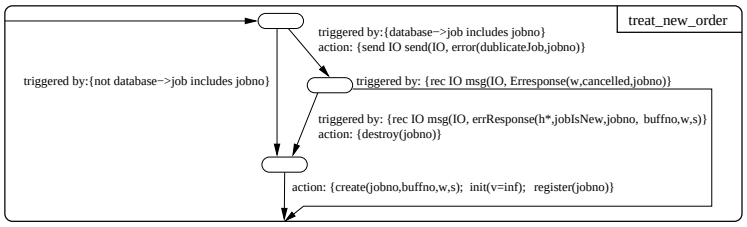
ior specification. It summarizes the treatment of job orders, missing and duplicate jobs and the deletion of jobs in individual sub statecharts, leading to a clear structuring of the behavior of the database. We could have refined the database capsule further by separating the maintenance of status information of the HTS and of the job queue by isolating them in an individual capsule of by introducing an uniform communication protocol. Since we focus on the application of the generation algorithm from SDs to statecharts here, we do not discuss further refinements in detail.

Also, the sub statecharts the database can simplified significantly by optimizations and refinements: The sub statechart `treat_new_order` in Figure 47(b) specifies the treatment of orders in a more compact manner than Figure 46 by applying the optimizations from Section 8.1.1, Figure 36. Again, first a refinement step has been carried out: joining the two separate paths from Figure 46 which create a new job capsule into a single path before combining the involved interactions into a single transition.

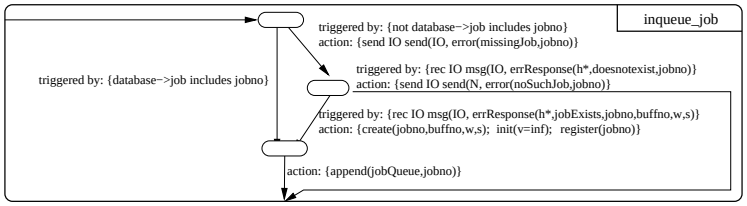
The sub statechart `delete_finished_job` in Figure 47(c) shows an important advantage of the use of combining the strengths of SDs and statecharts in the development process using our generation algorithm:



(a) Toplevel statechart



(b) Substatechart treat_new_order



(c) Substatechart inqueue_job

Figure 47: Toplevel statechart for the database

The scenarios developed in the previous sections do not cover the case when a job to be deleted does no longer exist due to previous communication errors similar to the scenarios considered for duplicate and missing jobs. Such incompletenesses can be easily detected in statechart specifications and fixed by specifying appropriate alternatives as shown in Fig. 47(c).

Of course, our model is also far from being complete: We haven't covered a lot of errors so far, and we could also think of extending the state model to further data structures such as the job queue (e.g. empty, nonempty). However, in this report, we are in conceptual models. Therefore, we leave such refinements of the model to further development steps transforming this model into an implementation.

8.4.2 A State Model for Jobs

The generation of a statechart for the capsule `Job` uses the sequence diagrams `DBNegotiationOfOrder`, `DBtakesWPintern`, `DBreleasesWPintern` and also the sequence diagrams which deal with error handling and create or destroy jobs. Again, we have specified no state information on the lifelines of the capsule job in the named sequence diagrams so far. The projections of the SDs with added state conditions are shown in Figure 48. For short, the figure shows only a subset of the scenarios dealing with the handling of errors.

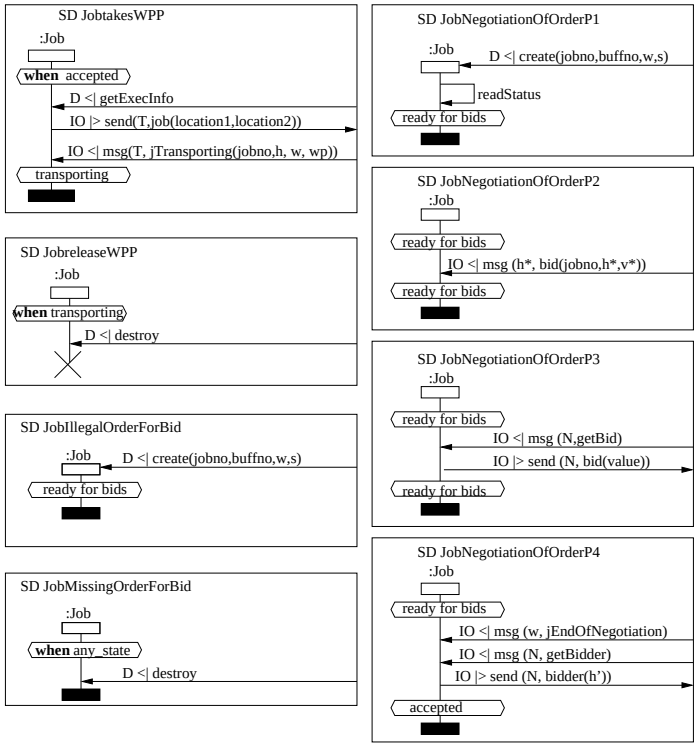


Figure 48: Normalized sequence diagrams for the capsule job

In the modeling approach taken in Figure 48, the state model of the capsule `job` follows closely the states of the actual job which is negotiated and executed. This design decision follows a common design pattern to concentrate the information model of a task which has to be carried out by the system into a separate capsule class. The start and termination of computations of `job` need not to be specified by conditions since they are implicitly specified by creation and destruction of the capsule class.

Still, we need to pay attention to the interrelation with the error handling. When a job is created during an error recovery procedure, the new created job needs to be put in the same state as its counterparts in other databases since this state reflects the state of the negotiation or execution of the respective transport. The scenario `JobIllegalOrderForBidP1/2/3`

leads to the state `ready for bids` – the same start as the regular creation of the job. The destruction of jobs in the course of an error handling procedure can occur at any time during the lifetime of a job. This is expressed by the condition `any_state` in the SD `JobMissingOrderForBid`. Applying the generation algorithm on these scenarios, we obtain the statechart shown in Figure 49.

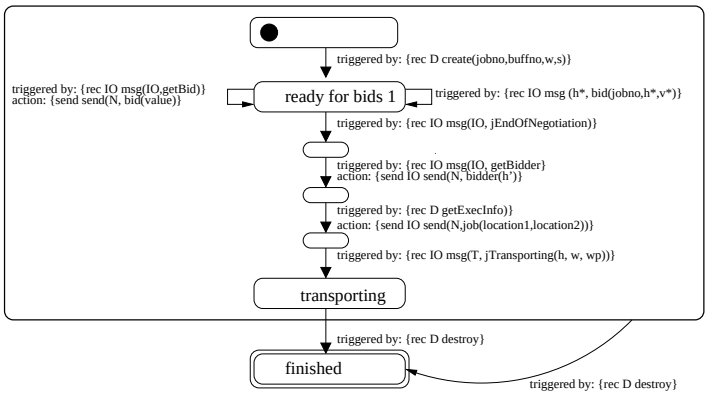


Figure 49: A statechart for the capsule job

Again, this statechart is only a prototype which represents the starting point of the design process. For example, the ordering information in the job statechart can be used to detect additional protocol errors, e.g. detecting `jTransporting` messages before the end of the negotiation. Such extensions can be added in the course of subsequent development steps.

8.5 The IOSystem

The remaining capsule within the HTS is IOSystem. The IOSystem is involved in almost any communication but - analogous to the database - it does not care about the order of the messages, it just forwards them to the addressee. Since we didn't add any scenarios which require state information, the statechart of the IOSystem is trivial: One state and one looping transition for each message type. Nevertheless, it is possible to develop useful statecharts for this capsule: For example, a protocol could be added which allows the sub capsules of the HTS system to notify the IOSystem about messages in which it is interested, to allow a flexible message forwarding. Further, we could think of recovery strategies for communication errors. These mechanisms can be introduced by applying interface refinement. Thus, the generation of a statechart for the IOSystem capsule is not interesting at the stage of development considered in this report but only at a more implementation oriented stage of development. Since technical modeling and interface refinement is out of scope of this report, the statechart is omitted here.

9 Conclusion

We have presented an approach at incorporating broadcast communication into the modeling of architectural design using UML-RT. We have shown that by means of only few syntactic extensions we can employ the UML's sequence diagrams for transparently capturing broadcasting scenarios. This enables concentrating on use cases and service-oriented specification techniques also in the development process for broadcasting systems. Making these extensions an integral part of the language allows both for a compact modeling of broadcasting and for adopting these aspects systematically into the systems architecture. Furthermore, integration of timing aspects, such as the durations of communications, can be easily integrated along the lines of what is already available for regular sequence diagrams in the UML [Rat98]. To cope with the complexity of "real" systems, notational extensions taken from message sequence charts have proven to be useful; especially the concept of condition labels allows to split complex scenarios into manageable parts. With these extensions, sequence diagrams can be used to describe interaction scenarios at any level of detail, although the notion of refinement of scenarios can only be mimicked within their syntax; yet there is no immediate support for their refinement.

To model broadcasting at the level of architecture, we have introduced and employed a pattern for capturing broadcasting by means of explicit components on all levels of the component hierarchy. This introduces broadcasting seamlessly on the basis of UML-RT's binary communication model. Dealing with broadcasting explicitly on the level of a logical architecture of the system under consideration has several advantages. It supports classical top-down structural system decomposition, and introduces a flexible, adaptable, and configurable communication mechanism we can exploit during further stages of requirements analysis and specification.

Using the algorithm developed in Section 5.2 and a the algorithm presented in [KGSB99], a prototypical architecture can be generated automatically from broadcasting scenarios captured by means of SDs. The resulting model includes all essential architectural aspects: component structures, component interfaces, and communication structures are described by capsule diagrams and protocols, and component behavior is described by UML-RT statecharts. The generated diagrams provide a high level architecture 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 level of detail at which a prototype of the systems architecture is generated can be freely chosen by the developers: As we have shown, exploring interaction requirements and developing the systems structure is an iterative process, and the generation of an architecture can be applied at an abstract level as well as at the level of refined scenarios. 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]).

With UML-RT, we have employed a powerful and widely used modeling language to demonstrate the benefits of our approach. Yet, the basic concepts are not limited by dependencies on specifics of the language: Architectural description techniques such as provided by UML-RT will also be essential improvements of the standard UML in its version 2.0. Our approach has the potential to support the integration of flexible communication regimes beyond broadcasting into arbitrary software architecture descriptions. Incorporating this support as a general design principle into corresponding case tools, which includes the integration of notational extensions into a formal semantics of UML-RT, is a necessary and promising area of further development of our approach.

Acknowledgments

The authors are grateful to Manfred Broy for encouraging and to Victoria Cengarle for reading draft versions and for her detailed comments.

References

[Bee94] Michael von der Beeck. A Comparison of Statecharts Variants. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Proc. Formal Techniques in Real-Time and Fault-Tolearnt Systems (FTRTFT'94)*, volume 863 of *LNCS*, pages 128–148. Springer, 1994.

[Dou98] Bruce Powel Douglass. *Real-time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley, 1998.

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

[fPuAI99] Fraunhofer-Institut für Produktionstechnik und Automatisierung IPA. Referenzfallstudie Produktionstechnik v1.3. `http://tfs.cs.tu-berlin.de/SPP/RefPAv13.ps`, 1999.

[Gmb02] ETAS GmbH. ASCET-SD product info. `http://www.etas.info/html/products/ec/ascetsd/en_products_ec_ascetsd_in%dex.php`, 2002.

[HP98] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts. The* STATEMATE *approach*. McGraw-Hill, 1998.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Series in Computer Science. Addison-Wesley, 1979.

[Inc02] The MathWorks Inc. The MathWorks product family. `http://www.mathworks.com/products/prodoverview.shtml`, 2002.

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

[IT99] ITU-T. *Z.120(11/99) – MSC2000*. ITU-T, Geneva, 1999.

[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.

[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.

[RS01] B. Rumpe and R. Sandner. UML - Unified Modeling Language im Einsatz. *at - Automatisierungstechnik, Reihe Theorie für den Anwender*, 49(8 - 10), 2001.

[San00] Robert Sandner. Developing Distributed Systems Step by Step with UML-RT. In Holger Giehse and Stephan Philippi, editors, *Workshop Visuelle Verhaltensmodellierung verteilter und nebenläufiger Software-Systeme*, pages 43–50. Universität Münster, 2000.

[Sel98] Bran Selic. Recursive Control. In Robert Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*, Software Patterns Series. Addison-Wesley, 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.