A CSP View on UML-RT structure diagrams

Clemens Fischer, Ernst-Rüdiger Olderog and Heike Wehrheim

Universität Oldenburg
Fachbereich Informatik
Postfach 2503, D–26111 Oldenburg, Germany
Fax: +49 441 7982965
{fischer,olderog,wehrheim}@informatik.uni-oldenburg.de

Abstract. UML-RT is an extension of UML for modelling embedded reactive and real-time software systems. Its particular focus lies on system descriptions on the *architectural* level, defining the overall system structure. In this paper we propose to use UML-RT structure diagrams together with the formal method CSP-OZ combining CSP and Object-Z. While CSP-OZ is used for specifying the system components themselves (by CSP-OZ classes), UML-RT diagrams provide the architecture description. Thus the usual architecture specification in terms of the CSP operators parallel composition, renaming and hiding is replaced by a *graphical* description. To preserve the formal semantics of CSP-OZ specifications, we develop a translation from UML-RT structure diagrams to CSP. Besides achieving a more easily accessible, graphical architecture modelling for CSP-OZ, we thus also give a semantics to UML-RT structure diagrams.

1 Introduction

Graphical modelling notations are becoming increasingly important in the design of industrial software systems. The Unified Modelling Language (UML [BRJ99,Obj99]), being standardised by the Object Management Group OMG, is the most prominent member of a number of graphical modelling notations for object-oriented analysis and design. UML-RT [SR98] is a UML profile proposed as a modelling language for embedded real-time software systems. Although the name RT refers to real-time, UML-RT's main extension concerns facilities for describing the *architecture* of distributed interconnected systems. UML-RT defines three new constructs for modelling structure: *capsules, ports* and *connectors*, and employs these constructs within UML's collaboration diagrams to obtain an architecture description. The advantage of UML-RT, like UML, is the *graphical* representation of the modelled system. However, it lacks a precise semantics.

A different approach to the specification of software systems is taken when a formal method is used as a modelling language. In contrast to UML, formal methods have a precise semantics, but mostly do not offer graphical means of specification. A joint usage of formal methods and graphical modelling languages could thus benefit from the advantages and overcome the deficiencies of each method. A number of proposals for combining UML with a formal method have already been made (e.g. [CE98,DLC98,KC99,OR00]). This paper makes another contribution in this field, focusing on one particular aspect of system modelling, the *architecture descriptions.*

The formal method we employ is CSP-OZ [Fis97,Fis00], a combination of the process algebra CSP [Hoa85,Ros97] and the specification language Object-Z [Smi00,Spi92]. The work presented in this paper can be seen as a first step towards an integration of UML and CSP-OZ. CSP-OZ has several features which makes it a suitable candidate for a formal method supporting UML. To name just two: it is an object-oriented notation (with concepts like classes, instantiation and inheritance), and, like UML, it combines a formalism for describing static aspects (Z) with one for describing the dynamic behaviour (CSP).

CSP-OZ specifications typically consist of three parts: first, some basic definitions of e.g. types are made; second, *classes* are defined, and finally, the *system architecture* (components and their interconnections) is fixed. All ingredients which usually appear in UML class descriptions can be found in CSP-OZ classes: attributes, methods and inherited superclasses are declared, associations can be modelled by using attributes with type of another class. Furthermore, one part of a CSP-OZ class specifies the *dynamic behaviour* of the class, which, in UML, is usually given by a separate diagram, e.g. a state chart. In contrast to UML, CSP-OZ uses the CSP process-algebraic notation for this purpose. The system architecture is given by instantiating the classes into a suitable number of objects and combining them using the CSP operators for parallel composition, hiding and renaming. To clarify this overall structure of the system, often some sort of ad-hoc connection diagram is drawn. But these diagrams only serve as an illustration of the CSP architecture description; neither is the form of the diagrams fixed in any way, nor do they have a formal semantics. Hence they cannot actually *replace* the CSP description.

For the integration of CSP-OZ and UML we start here with this last part of CSP-OZ specifications. Defining the system architecture in the above described sense is exactly the intended purpose of UML-RT structure diagrams. Our proposal in this paper is therefore to replace the textual CSP architecture descriptions by UML-RT structure diagrams. To preserve the precision of a formal method we fix the syntax and semantics of these diagrams. The advantages are twofold: UML-RT provides us with a widely accepted graphical specification technique for defining architectures, and additionally a formal semantics for UML-RT structure diagrams in the setting of distributed communicating systems is achieved. For the other main ingredients of CSP-OZ specifications, the classes, we envisage an integration with UML in the following way: CSP-OZ classes are split into a static part, with an appropriate representation by UML class diagrams, and a dynamic part, with a representation by e.g. an activity diagram or a state chart. However, in this paper we are only concerned with obtaining a graphical description of the architecture specification.

Technically, the use of UML-RT diagrams within CSP-OZ is achieved by the following two issues: a formalisation of the syntax of UML-RT diagrams with Object-Z and a formalisation of their semantics using the CSP operators for parallel composition, renaming and hiding. The major difficulty in giving a semantics to UML-RT diagrams is the treatment of the *multi-object* notation of UML: a component may communicate with a multi-object (a number of instances of the same class) over a single channel. In this case some kind of *addressing* has to be introduced to achieve communication with a particular instance, i.e. the channel additionally has to carry addresses of receivers. Since the need for introducing an addressing mechanism in communication is already visible in the architecture description, the translation also has to take addressing into account. This is solved by introducing special address parameters for processes and carrying out appropriate renamings on channel names of components.

The case study in Section 4 demonstrates the usefulness of the multi-object notation in a real-life application. In our applications the basic capsules in UML-RT structure diagrams stand for CSP-OZ classes or simply CSP processes. However, in this paper the details of CSP-OZ are not important. Here we only need the architectural operators of CSP that allow us to connect CSP-OZ classes or CSP processes: parallel composition, renaming and hiding.

For UML-RT the results of this paper are a contribution to its formal semantics. We nevertheless believe that this is not the only possible interpretation (in particular since we only use a very simple form of protocols allowing for synchronous communication over one channel); in a different setting a different semantics might be conceivable. For CSP and CSP-OZ the benefit is a precise type of diagram for describing the system architecture, replacing the informal connection diagrams that appear in books on process algebra like [Hoa85], viz. UML-RT structure diagrams. Particularly interesting is the use of multi-objects of UML-RT for a concise description of iterated CSP operators.

The paper is organised as follows. The next section gives a short introduction to the specific constructs of UML-RT and formalises the syntax of structure diagrams using Object-Z. Section 3 defines the semantics of these diagrams by a translation to CSP. We illustrate our approach by some smaller examples and, in Section 4, by the case study of an automatic manufacturing system. The conclusion summarises our work and discusses some related approaches.

2 UML-RT

Currently, the UML, as standardised by the OMG, is the most widely used object-oriented modelling language. UML-RT (UML for Real-Time Systems) [SR98] is an extension of UML designed for describing *architectures* of embedded real-time systems. The emphasis of this extension lies on the modelling of the *structure* of distributed systems, no particular real-time features are added to UML's possibilities. The extension uses standard UML tailoring mechanisms like stereotypes and tagged values.

In the following, we briefly describe UML-RT and give some small examples. UML-RT defines three constructs for modelling the structure of distributed systems:

Capsules Capsules describe complex components of systems that may interact with their environment. Capsules may be hierarchically structured, enclosing a number of subcapsules which themselves may contain subcapsules.

Ports Interaction of capsules with the environment is managed by attaching ports to capsules. They are the only means of interaction with the environment. Ports are often associated with *protocols* that regulate the flow of information passing through a port. Ports can furthermore be either *public* or *private*. Public ports have to be located on the border of a capsule.

Connectors Connectors are used to interconnect two or more ports of capsules and thus describe the communication relationships between capsules.

For all three constructs stereotypes are introduced: ≪capsule≫ and ≪port≫ are stereotypes for particular classes, whereas ≪connector≫ is a stereotype used for an association. The ports of a capsule class are listed in a separate compartment after the attribute and operator list compartments. A class diagram can be used to define all capsule classes of a system. The actual architecture of the system is given by a *collaboration diagram*, fixing the components of the system and their interconnections. In a collaboration diagram objects (instances of capsule or ports classes) are represented by *capsule roles* and *port roles*. Port roles are indicated by small black- or white-filled squares. Figure 1 gives a first example of an UML-RT collaboration diagram.

The collaboration diagram shows a capsule of class *CapsA* consisting of two subcapsules *CapsX* and *CapsY*. These capsules have ports $p1$ and $q1$, which are connected and private within capsule *CapsA*. Capsule *CapsY* furthermore has a port $p2$ connected with the public port $q2$ of capsule *CapsA*. Capsule *CapsA* is connected with capsule *CapsB* via a connector $c3$ between public ports $q2$ and $r1$.

In the case of binary protocols (two participants in the communication), a port and its counterpart can be depicted by black- and white-filled squares, respectively.

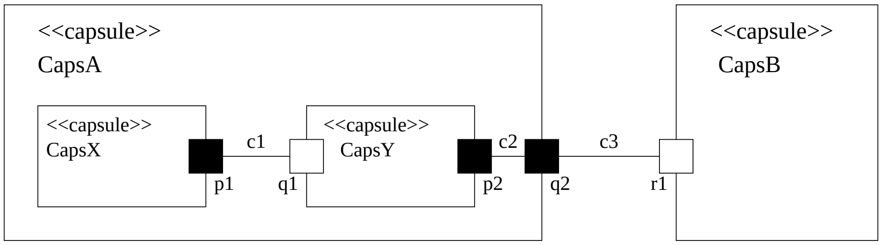

Fig. 1. Collaboration diagram

In the following we will always use the black-filled port for the sender and the white-filled port for the receiver of messages.

Another UML notation frequently used in UML-RT collaboration diagrams is the *multi-object* notation (depicted as a stack of rectangles); a number may be used to indicate the actual number of instances. Figure 2 shows the use of the multi-object notation for capsules and ports. Capsule A can communicate with instances of the multi-capsule B via port p. The instance names $\{u, v, w\}$, attached to the multi-capsule via a note, can be used for addressing the desired communication partner. Addressing is needed since there is a single name for the sending port but multiple receivers. The multiplicity of the sending port p indicates this addressing.

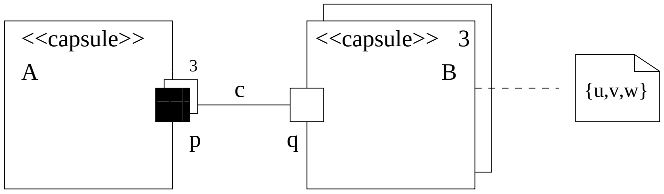

Fig. 2. Multi-object notation for capsules

2.1 Z formalisation of the syntax of UML-RT diagrams

For the translation of UML-RT collaboration diagrams into CSP we need a precise description of their syntax. To this end, we use the formal method Z and its extension Object-Z, which are frequently employed for syntax descriptions of UML diagrams [FR99,KC99].

The building blocks of UML-RT collaboration diagrams are ports, connectors, capsules and instances of capsules. For each of these we need a type for their names:

$$[PortName, ConName, CapName, InstName]$$

A *port* is always connected to some capsule. Thus a port has a port name, a capsule name and a *colour*, viz. black or white. The intuition is that input ports are white and output ports are black.

$$Colour ::= black \mid white$$

Furthermore, a port may be a multi-object and thus has a multiplicity specified. This is represented as an Object-Z class

Port

$name : PortName$
$cname : CapName$
$colour : Colour$
$multi : \mathbb{N}_1$

$colour = white \Rightarrow multi = 1$

with a state consisting of the attributes *cname, colour* and *multi*. We thus assume that white ports (the receivers) always have multiplicity one. The particular receiver of a message is solely determined by the sender. A port with multiplicity 1 is called *single port* and a port with multiplicity greater than 1 is called a *multi-port*.

We assume that port names are unique, i.e. there exists an injective function $\mathcal{P}$ from port names to ports:

$\mathcal{P} : PortName \rightarrowtail\!\!\!\!\!\!+\;\; Port$

$\forall\, na : \mathrm{dom}\,\mathcal{P} \bullet na = \mathcal{P}(na).name$ [$\mathcal{P}$ respects port names]

The function $\mathcal{P}$ is a partial function because not every port name has to be used in a given design.

A *connector* comprises a name and a finite set of ports. For simplicity we assume that every connector combines exactly two ports. Thus we only consider the case of *binary protocols* or point-to-point communication. The name of a connector is not always given in UML diagrams, but we assume that some unique default name for every connector can be generated. This is represented by an Object-Z class

Connector

$name : ConName$
$ports : \mathbb{F}\, Port$

$\#ports = 2$

where the state has the attributes *name* and *ports*. Also connector names are unique, which is formalised by a suitable injective function $\mathcal{N}$:

$\mathcal{N} : ConName \rightarrowtail\!\!\!\!\!\!+\;\; Connector$

$\forall\, na : \mathrm{dom}\,\mathcal{N} \bullet na = \mathcal{N}(na).name$ [$\mathcal{N}$ respects capsule names]

In UML-RT system components are represented by capsules. We distinguish between *basic, compound* and *multi-object capsules*. This is formalised using Object-Z by defining a base class Capsule which is then extended via inheritance to subclasses CompCapsule and MultiCapsule. A basic capsule will be translated into a CSP-OZ class or simply a CSP process. It usually has a number of ports to be able to communicate with its environment. A compound capsule has in addition some subcapsules linked by connectors. A multi-object capsule is obtained from a basic or compound capsule by adding a multiplicity.

Each capsule has a name. As before we assume that this name is unique, i.e. there exists an injective function $\mathcal{C}$ from capsules names to capsules (and all their subclasses denoted by the operator $\downarrow$):

$\mathcal{C} : \mathit{CapName} \rightarrowtail\!\!\!\!\!\!+\;\; \downarrow \mathit{Capsule}$ [Capsule and all its subclasses]

$\forall\, na : \operatorname{dom} \mathcal{C} \bullet na = \mathcal{C}(na).name$ [$\mathcal{C}$ respects capsule names]

A *capsule* has a name and a finite set of ports that are considered as *public*, i.e. accessible from the environment. This is represented as an Object-Z base class where the state has the attributes *name* and *ports*:

Capsule

$$name : CapName$$
$$ports : \mathbb{F}\, Port$$
$$\forall\, p : ports \bullet p.cname = name$$

The consistency condition requires that all ports of a capsule refer to that capsule.

A *compound capsule* extends a given capsule by an inner structure consisting of a finite set of subcapsules, referenced by names, and a finite set of connectors. We use therefore the inheritance notation of Object-Z [1]:

CompCapsule

inherit *Capsule*

$scnames : \mathbb{F}\, CapName$ [names of subcapsules]

$conn : \mathbb{F}\, Connector$ [inner connectors]

$$\forall\, c : conn \bullet c.ports \subseteq ports \cup \bigcup\{sn : scnames \bullet \mathcal{C}(sn).ports\} \qquad [1]$$
$$\forall\, c_1, c_2 : conn \bullet c_1.name = c_2.name \Rightarrow c_1 = c_2 \qquad [2]$$
$$\forall\, c_1, c_2 : conn \bullet c_1.ports \cap c_2.ports \neq \varnothing \Rightarrow c_1 = c_2 \qquad [3]$$
$$\forall\, c : conn \bullet \forall\, p, p' : c.ports \bullet p \neq p' \Rightarrow p.cname \neq p'.cname \qquad [4]$$
$$\begin{array}{l} \forall\, c : conn \bullet \forall\, p, p' : c.ports \bullet (\ p \neq p' \Rightarrow \\ \quad (p \in ports \Rightarrow p.multi = p'.multi \land p.colour = p'.colour) \qquad [5] \\ \qquad \land \\ \quad (p, p' \notin ports \Rightarrow p.colour \neq p'.colour \land \qquad [6] \\ \qquad ((p.multi = 1 \land p'.multi = \mathcal{C}(p.cname).multi) \\ \qquad\quad \lor \\ \qquad (p'.multi = 1 \land p.multi = \mathcal{C}(p'.cname).multi))\ )\ ) \end{array}$$

The ports of the subcapsules are treated as *private*, i.e. hidden form the environment. The connectors link these private ports of the subcapsules to each other or to the public ports of the whole capsule. The predicates state consistency conditions for compound capsules:

- connectors can only connect ports given in the capsule (condition 1),
- the name of a connector is unique within a capsule (condition 2),
- any two connectors with a common port are identical, i.e. connectors represent point-to-point connections without fan-out (condition 3),
- connectors may only connect ports of different capsules (condition 4).

Furthermore the multiplicities of the ports of a connector have to match:

[1] Inheritance in Object-Z is syntactically expressed by simple inclusion of a class. Semantically inheritance is expressed by signature extension and logical conjunction.

- if one port of the connector is public, the multiplicities and colours of both ports of the connector coincide (condition 5),
- if both ports of the connector are private, the colours of the ports differ, one is a single port and the other one has a multiplicity that coincides with the multiplicity of the subcapsule of the single port (condition 6, see also Figure 2). Note that this covers the case that both ports and hence both their subcapsules are single ones.

A *multi-capsule* extends a basic or compound capsule by a multiplicity and the names of the instances (we assume that these names are either given in the diagram, e.g. by attaching a note to the multi-capsule, or can be generated). Using inheritance this can be represented as follows:

MultiCapsule

inherit *Capsule*

$scname : CapName$
$multi : \mathbb{N}_1$
$inames : \mathbb{F}\ InstName$

$\#inames = multi$

Starting from ports, connectors and capsules, systems can be built up. A *system* is defined as an outermost capsule.

3 Translating UML-RT diagrams to CSP

UML-RT structure diagrams give a graphical description of the architecture of systems, their components and interconnections. The Object-Z formalisation so far fixes the valid syntactic structures of these diagrams. For giving a formal semantics of the behavioural aspects of the diagrams, we use another formal method which is particularly suited for the description of distributed communicating systems: the process algebra CSP [Hoa85,Ros97].

3.1 A brief re-cap of CSP

CSP is a formal method for specifying and reasoning about processes. CSP describes a system as a number of processes, possibly running in parallel and synchronously communicating over named *channels*. Each process is built over some set of communication events, using operators like sequential composition or choice to construct more complex processes. For describing the *architecture* of systems the operators *parallel composition*, *hiding* and *renaming* are used.

Parallel composition Parallel composition, denoted by $||_A$, is used to set processes in parallel, requiring synchronisation on all communication events in the set A. A communication event consists of a channel name and some values of parameters (e.g. $ch.v_1.v_2$). For instance, the term $C_1 \ ||_{\{ch.1, ch.2\}} \ C_2$ describes a parallel composition of components C_1 and C_2 with joint execution of events $ch.1$ and $ch.2$. Often the synchronisation set is simply a set of channels and then stands for synchronisation on all events built over these channel names.
The operator $|||$ stands for *interleaving*, i.e. parallel composition with empty synchronisation set. Since interleaving is associative, it can be iterated:

$$|||_{i:I}\, P_i \qquad \text{where } I \text{ is a finite index set}$$

Alphabetised parallel $P \,_A\|_B\, Q$ is another version of parallel composition. If A and B are sets of events, $P \,_A\|_B\, Q$ is the combination where P is allowed to communicate in the set A, called the alphabet of P, and Q is allowed in the set B, the alphabet of Q, and both P and Q must agree on events in the intersection $A \cap B$. Also alphabetised parallel can be iterated. For a finite index set I the notation is

$$\Big\|_{i:I}[A_i] \bullet P_i \qquad \text{where } A_i \text{ is the alphabet of } P_i.$$

Hiding Hiding, denoted by $\backslash A$, is used to make communication events *internal* to some components. Technically, hiding is achieved by renaming some events into the invisible event τ. Thus they are not available for communication anymore. This corresponds well to the concept of private ports.

Renaming Renaming, denoted by $[R]$ where R is a relation between events, is used to rename communication events. The renaming most often only concerns the channel names, not the values of parameters. Therefore, R may also be a relation on channel names. For instance, the term $C[in \mapsto out]$ describes a process C where all communication events of the form $in.x$ are renamed into $out.x$.

3.2 Translation of examples

In general, every capsule of a diagram stands for an instantiation of a specific CSP process. Ports are modelled by channels, subcapsules within some capsule have to be put into parallel composition with appropriate synchronisation sets guaranteeing the interconnections among capsules as defined by the connectors. If a capsule is basic and contains no further subcapsules, the concrete CSP process remains undefined (since collaboration diagrams do not model the precise behaviour of components, only the structure of the system) and the only part that is used in the architecture description is the name of the capsule. The actual class or process definition behind the capsule has to be specified somewhere else. Given the process names for these basic non-hierarchical capsules, the CSP term for the whole diagram can be inductively constructed.

Before giving a formal definition of the translation of UML-RT diagrams into CSP, we explain the translation informally using the two examples shown in Figures 1 and 2 of the last section. In Figure 1, there are three basic capsules which do not contain any subcapsules. For these three we assume to have some definition at hand (e.g. as a CSP-OZ class) and just use their names:

$$CapsX, CapsY, CapsB$$

All ports of the three capsules have multiplicity 1. Therefore no addressing is needed here and the process names are not parametrised. For deriving the process term of the compound capsule $CapsA$, we now have to compose processes $CapsX$ and $CapsY$ in parallel. The choice of the synchronisation set requires some care. If we simply use the names p_1 and q_1 of the ports attached to the connector c_1, no communication is possible at all in the CSP model because $CapsX$ communicates only on channel p_1 whereas $CapsY$ communicates only on channel q_1. Instead we use the name of the *connector* in the synchronisation set and carry out an appropriate renaming of port names to connector names on the subcapsules by $R_X = \{p_1 \mapsto c_1\}$ and $R_Y = \{q_1 \mapsto c_1, p_2 \mapsto c_2\}$:

$$CapsX[R_X] \;_{\{c_1\}}\|_{\{c_1,c_2\}}\; CapsY[R_Y]$$

Thus communication between the capsules $CapsX$ and $CapsY$ is modelled by the CSP paradigm of synchronous (or handshake or rendezvous) communication.

For obtaining the process for the compound capsule *CapsA* two more operations have to be applied: the channel c_2 has to be renamed into q_2 (the name of the public port on the border of *CapsA*) and afterwards the channel c_1 (connector of two private ports) has to be hidden. Summarising, we get the following CSP process for capsule *CapsA*:

$$CapsA = \big(CapsX[R_X]\ _{\{c_1\}}||_{\{c_1,c_2\}}\ CapsY[R_Y]\big)[c_2 \mapsto q_2] \setminus \{c_1\}$$

The CSP process describing the complete architecture of the system is then the parallel composition of the two capsules *CapsA* and *CapsB*, again applying renaming, synchronisation and hiding of private ports. *System* can be seen as the compound capsule containing all capsules in the diagram.

$$System = \big(CapsA[q_2 \mapsto c_3]\ _{\{c_3\}}||_{\{c_3\}}\ CapsB[r_1 \mapsto c_3]\big) \setminus \{c_3\}$$

The example in Figure 2 requires a careful treatment of addressing. Port p with multiplicity 3 indicates the possibility of capsule A to choose between different receivers of messages sent over p. This is modelled by parameterising the process name of A with a formal parameter Adr_p standing for the set of possible receivers: $A(Adr_p)$. For example, the CSP process

$$A(Adr_p) = Produce(e);\ (\ |||_{out:Adr_p}\ p.out!e);\ A(Adr_p)$$

repeatedly produces an element e which is then output in parallel to all addresses *out* of the set Adr_p via the multi-port p. Instantiating the formal parameter Adr_p with the address set $\{u, v, w\}$ yields

$$A(\{u, v, w\}) = Produce(e);\ (p.u!e\ |||\ p.v!e\ |||\ p.w!e);\ A(\{u, v, w\}).$$

Basic capsule B is treated as before. For example, the CSP process

$$B = q?x;\ Consume(x);\ B$$

repeatedly receives an element along port q, stores it in a local variable x and consumes it.

Next, the process for the multi-capsule has to be constructed. Let us name this process *MB* (for *MultiB*). The semantics of a multi-capsule is the interleaving of all its instances. To achieve a correct addressing of the instances, communication over port q in the instance *in* is renamed into communication over $q.in$:

$$MB =\ |||_{in:\{u,v,w\}}\ B[q \mapsto q.in]$$

Thus the instance name *in* is transmitted as part of the value over channel q. For the above example B we obtain

$$B[q \mapsto q.in] = q.in?x;\ Consume(x);\ B[q \mapsto q.in].$$

Finally, compound capsule *System* is constructed. This requires an instantiation of the address parameter of A with the set of receivers $\{u, v, w\}$, the instance names of the multi-capsule attached to port p.

$$System = \big(A(\{u, v, w\})[p \mapsto c]\ _{\{c\}}||_{\{c\}}\ MB[q \mapsto c]\big) \setminus \{c\}$$

Capsule A may now use the instance names u, v, w as parameters for channel p and thus send messages to particular instances.

3.3 Translation in general

Now we present a function $\mathcal{T}$ for translating a given capsule into a CSP process in equational form. We start from the basic types

$$[\mathit{Process}, \mathit{Chans}, \mathit{Val}]$$

of CSP processes, CSP channels, and values sent along channels. We assume a structure on the set of *Events* based on a recursive free type *Data*:

$$\begin{array}{l} \mathit{Data} ::= \mathit{basic}\langle\!\langle \mathit{Val} \rangle\!\rangle \mid \mathit{comp}\langle\!\langle \mathit{InstName} \times \mathit{Data} \rangle\!\rangle \\ \mathit{Events} == \mathit{Chans} \times \mathit{Data} \end{array}$$

By convention, we abbreviate data of the form $\mathit{comp}(\mathit{in}, v)$ by $\mathit{in}.v$ and data of the form $\mathit{basic}(v)$ by just v. Also, following CSP conventions, events of the form (ch, d) are written as $\mathit{ch}.d$ so that a typical event will appear as $\mathit{ch}.\mathit{in}_1.....\mathit{in}_m.v$ where $m \in \mathbb{N}$. This dot notation for CSP communications should not be confused with the selection of components in Z schemas. Here the sequence $\mathit{in}_1.....\mathit{in}_m$ of instance names will play the role of addresses to where the value v should be sent along the channel ch.

With every process there is an *alphabet* of events associated:

$$\mid \ \alpha : \mathit{Process} \rightarrow \mathbb{P}\,\mathit{Events}$$

The translation function

$$\mid \ \mathcal{T} : \mathit{Capsule} \rightarrow \mathit{ProcessEquations}$$

generates a set of process equations of the form

$$\mathit{name}(\mathit{ParameterList}) = \mathit{ProcessExpression}$$

This set is defined inductively on the syntactic structure of capsules.

(1) Let BC be a basic capsule with $BC.\mathit{ports} = \{p_1, ..., p_m, q_1, ..., q_n\}$ where $m, n \in \mathbb{N}$ and $p_1, ..., p_m$ are single ports, i.e. with $p_i.\mathit{multi} = 1$ for $i = 1, ...m$, and $q_1, ..., q_n$ are multi-ports, i.e. with $q_j.\mathit{multi} > 1$ for $j = 1, ..., n$. Then we take new formal parameters $\mathit{Adr}_1, ..., \mathit{Adr}_n$ standing for sets of instance names that will serve as addresses to which the multi-ports $q_1, ..., q_n$ can be connected.
The translation function $\mathcal{T}$ generates one process equation $\mathcal{T}(BC)$ for BC:

$$\mathcal{T}(BC) \equiv BC.\mathit{name}(\mathit{Adr}_1, ..., \mathit{Adr}_n) = \mathit{RHS}$$

Here $\equiv$ stands for syntactic identity and *RHS* for a process with alphabet

$$\begin{array}{l} \alpha(\mathit{RHS}) \subseteq (\{p_1, ..., p_m\} \times \mathit{Data}) \cup \\ \qquad\qquad (\{q_1\} \times \mathit{comp}(\mathit{Adr}_1 \times \mathit{Data}) \cup ... \cup \{q_n\} \times \mathit{comp}(\mathit{Adr}_n \times \mathit{Data})). \end{array}$$

In our setting, *RHS* is the name of the corresponding CSP-OZ class.

(2) Let CC be a compound capsule with $CC.\mathit{ports} = \{p_1, ..., p_m, q_1, ..., q_n\}$ where $m, n \in \mathbb{N}$ and $p_1, ..., p_m, q_1, ..., q_n$ are as above. Then we take new formal parameters $\mathit{Adr}_1, ..., \mathit{Adr}_n$ as above and define

$$\begin{array}{l} \mathcal{T}(CC) \equiv \\ \quad CC.\mathit{name}(\mathit{Adr}_1, ..., \mathit{Adr}_n) = \\ \qquad\quad \big((\ \|_{SN:CC.\mathit{scnames}} [\mathcal{C}(SN).\mathit{ports}[R_{SN,CC}]] \bullet \\ \qquad\qquad\qquad\qquad SN(B_1, ..., B_{k(SN)})[R_{SN,CC}])[P_{CC}]\big) \setminus H_{CC} \end{array} \qquad \text{[Eqn 1]}$$

$$\mathcal{T}(\mathcal{C}(SN)) \quad \text{for all } SN : CC.\mathit{scnames} \qquad \text{[Eqns 2]}$$

Thus $\mathcal{T}(CC)$ generates one new process equation (Eqn 1) where the right-hand side uses an iterated *alphabetised parallel* composition and adds to it the equations (Eqns 2) obtained by applying the translation function $\mathcal{T}$ inductively to all subcapsules in CC. The alphabetised parallel iterates over all names SN of subcapsules in CC.

Suppose $r_1, ..., r_{k(SN)}$ are the multi-ports of the subcapsule with the name SN. Then the actual address parameters B_i with $i = 1, ..., k(SN)$ for these multi-ports are defined as follows:

- $B_i = Adr_{p_j}$ if r_i is connected to a public multi-port p_j of CC
- $B_i = MC.inames$ if r_i is connected to a single port p_j of a multi-capsule MC inside CC

The renaming $R_{SN,CC}$ changes port names to connector names:

$$\begin{aligned} R_{SN,CC} = \{&pn : PortName;\ cn : ConName \mid \\ &\exists\, p : \mathcal{C}(SN).ports;\ c : CC.conn \bullet \\ &p \in c.ports \wedge pn = p.name \wedge cn = c.name \bullet (pn \mapsto cn)\} \end{aligned}$$

The renaming P_{CC} is used to rename connectors from a subcapsule to a public port on the border of the capsule back to the name of the public port:

$$\begin{aligned} P_{CC} = \{&cn : ConName;\ pn : PortName \mid \\ &\exists\, p : CC.ports;\ c : CC.conn \bullet \\ &p \in c.ports \wedge pn = p.name \wedge cn = c.name \bullet (cn \mapsto pn)\} \end{aligned}$$

Finally, all remaining connector names are hidden:

$$H_{CC} = \{c : CC.conn \bullet c.name\}$$

(3) Let MC be a multi-capsule with

$$\mathcal{C}(MC.cname).ports = \{p_1, ..., p_m, q_1, ..., q_n\}$$

where $m, n \in \mathbb{N}$ and $p_1, ..., p_m, q_1, ..., q_n$ are as above. Again we take new formal parameters $Adr_1, ..., Adr_n$ as above and define

$$\begin{aligned} &\mathcal{T}(MC) \equiv \\ &\quad MC.name(Adr_1, ..., Adr_n) = && \text{[Eqn 1]} \\ &\qquad\qquad \Big|\Big|\Big|_{in:MC.inames} MC.cname(Adr_1, ..., Adr_n)[\![in]\!] \\ &\quad \mathcal{T}(\mathcal{C}(MC.cname))) && \text{[Eqn 2]} \end{aligned}$$

Thus $\mathcal{T}(MC)$ generates one new process equation (Eqn 1) where the right-hand side uses an iterated *interleaving* operator and adds to it the equation (Eqn 2) obtained by applying the translation function $\mathcal{T}$ inductively to the capsule inside MC. The interleaving operator iterates over all instances of this capsule. Each *instance* is formalised by applying a renaming operator denoted by the postfix $\_[\![in]\!]$ where in is an instance name. For a given process P this operator is defined by

$$P[\![in]\!] \equiv P[\{ch : Chans;\ d : Data \mid ch.d \in \alpha(P) \bullet ch.d \mapsto ch.in.d\}]$$

(cf. the construction of the process MB in the example in subsection 3.2).

In the following we assume that a collaboration diagram implicitly contains a capsule *System* enclosing all capsules appearing in the diagram. The CSP process for the capsule *System* gives the architecture description.

4 Case Study: Automatic Manufacturing System

In this section we apply our approach to a larger case study. It concerns the architectural description of an *automatic manufacturing system*. In automatic manufacturing systems the transportation of material between machines is carried out by autonomous or *holonic* transportation agents, i.e. vehicles or robots without drivers and without a central control for scheduling[2]. In [Weh00] a CSP-OZ specification of an automatic manufacturing system is given in which the architecture of the system is described by a CSP term and the components (capsules) are given by CSP-OZ class definitions. Here, we will only present the architecture, initially modelled by a UML-RT collaboration diagram and then translated into a CSP term using the translation $\mathcal{T}$ of the previous section.

The automatic manufacturing system consists of the following parts (see Figure 3): two stores *In* and *Out*, one for workpieces to be processed (the in-store) and one for the finished workpieces (the out-store); two holonic transportation systems (Hts) $T1$ and $T2$; and three machines (Wzm[3]) A, B and C for processing the workpieces.

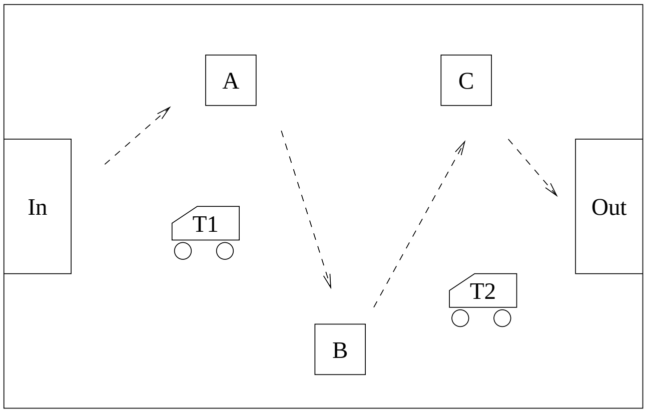

Fig. 3. Plant

Every workpiece has to be processed by all three machine tools in a fixed order ($In \rightarrow A \rightarrow B \rightarrow C \rightarrow Out$). The Hts' are responsible for transporting the workpieces between machines and stores. They work as autonomous agents, free to decide which machine to serve (within some chosen strategy). Initially the in-store is full and the out-store as well as all machines are empty. When a machine is empty or contains an already processed workpiece it broadcasts a *request* to the Hts in order to receive a new workpiece or to deliver one. The Hts' (when listening) send some *offer* to the machines, telling them their cost for satisfying the request. Upon receipt of offers the machine decides for the best offer and give this Hts the *order*, which then executes it. Execution of a job involves *loading* and *deloading* of workpieces from/to Hts and from/to stores and machines. This way, all workpieces are processed by all three tools and transported from the in- to the out-store.

The CSP-OZ specification of this manufacturing system contains class definitions *Store* and *Wzm*. The most complex component *Hts* is split into three parts: one

[2] This case study is part of the priority research program "Integration of specification techniques with applications in engineering" of the German Research Council (DFG) (`http://tfs.cs.tu-berlin.de/projekte/indspec/SPP/index.html`).

[3] in German: Werkzeugmaschine

for managing the acquisition of new jobs (*Acquisition*), one for coordinating the driving in the plant hall (*Driver*) and a control part (*HtsCtrl*). The control part calls component *Acquisition* when an order should be *acquired* (with response *new order*), and component *Driver* when the vehicle has to move. Furthermore, component *Acquisition* frequently asks *Driver* about their current position, which is influencing the cost of offers.

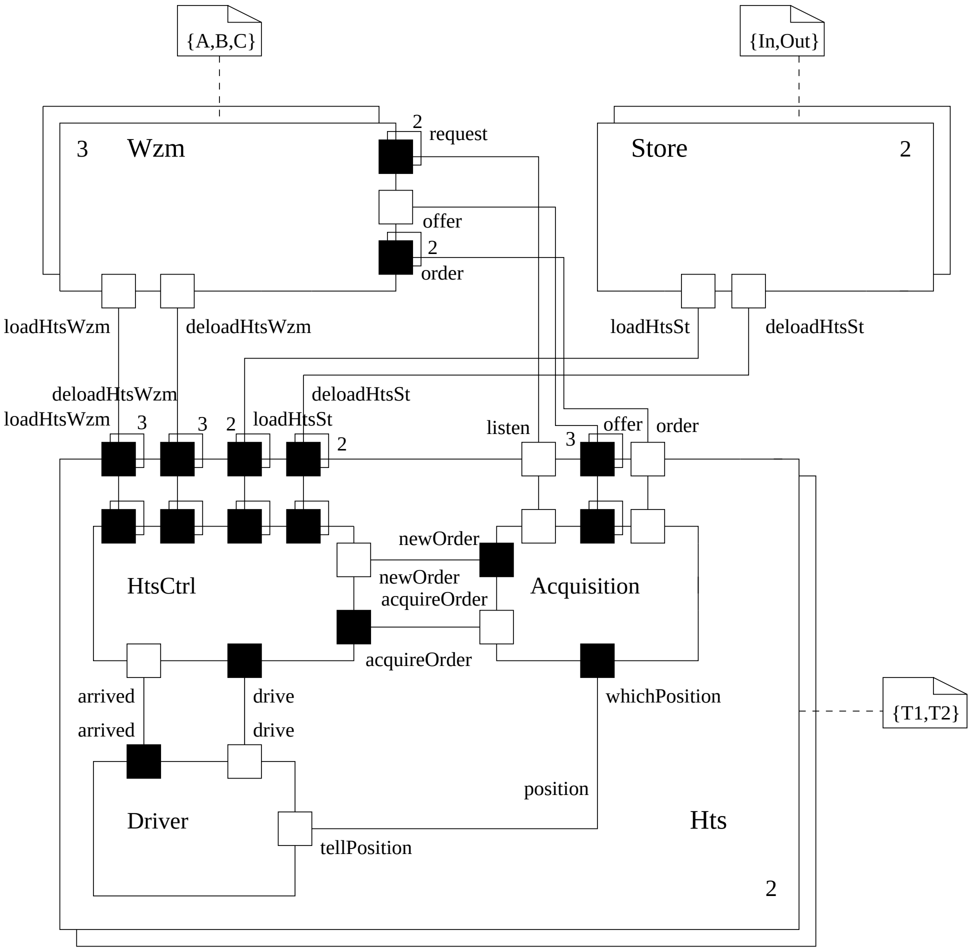

Fig. 4. Architecture of the Manufacturing System

Figure 4 shows the architecture of the manufacturing system as an UML-RT structure diagram. In most cases we have omitted the names of connectors. They only appear at the places where they are necessary in the transformation: when the connected ports have different names. In the other cases, we simply assume that the connector name equals the names of connected ports (and consequently omit the otherwise necessary renaming). We also sometimes omit port names of subcapsules when they agree with the port names of the compound capsule. The set of process equations corresponding to the graphical architecture description is constructed inductively. First, the process names and parameters for all basic capsules are fixed. For each of these names, a corresponding CSP-OZ class has to be declared in the rest of the specification. Thus we obtain:

$$\begin{array}{l} \mathit{Store} \\ \mathit{Driver} \\ \mathit{Acquisition}(\mathit{Adr}_{\mathit{offer}}) \\ \mathit{HtsCtrl}(\mathit{Adr}_{\mathit{loadHtsWzm}}, \mathit{Adr}_{\mathit{deloadHtsWzm}}, \mathit{Adr}_{\mathit{loadHtsSt}}, \mathit{Adr}_{\mathit{deloadHtsSt}}) \\ \mathit{Wzm}(\mathit{Adr}_{\mathit{request}}, \mathit{Adr}_{\mathit{order}}) \end{array}$$

Next, we construct the process equations for the multi-capsules *Store* and *Wzm*. Their process names are *MStore* and *MWzm*, respectively.

$$\mathit{MStore} = \mathit{Store}[\![\mathit{In}]\!] \;|||\; \mathit{Store}[\![\mathit{Out}]\!]$$

$$\begin{array}{l} \mathit{MWzm}(\mathit{Adr}_{\mathit{request}}, \mathit{Adr}_{\mathit{order}}) = \\ \qquad |||_{w:\{A,B,C\}}\ \mathit{Wzm}(\mathit{Adr}_{\mathit{request}}, \mathit{Adr}_{\mathit{order}})[\![w]\!] \end{array}$$

The component Hts is the most complex one. First, the process equation for the compound capsule is constructed.

$$\begin{array}{l} \mathit{Hts}(\mathit{Adr}_{\mathit{loadHtsWzm}}, \mathit{Adr}_{\mathit{deloadHtsWzm}}, \\ \qquad \mathit{Adr}_{\mathit{loadHtsSt}}, \mathit{Adr}_{\mathit{deloadHtsSt}}, \mathit{Adr}_{\mathit{offer}}) = \\ \quad \big(\mathit{Driver}[\mathit{tellPosition} \mapsto \mathit{position}]\ {}_{A_{\mathit{Driver}}}||_{A_{\mathit{Acqui}} \cup A_{\mathit{Ctrl}}} \\ \qquad \big(\mathit{Acquisition}(\mathit{Adr}_{\mathit{offer}})[\mathit{whichPosition} \mapsto \mathit{position}]\ {}_{A_{\mathit{Acqui}}}||_{A_{\mathit{Ctrl}}} \\ \qquad \mathit{HtsCtrl}(\mathit{Adr}_{\mathit{loadHtsWzm}}, \mathit{Adr}_{\mathit{deloadHtsWzm}}, \mathit{Adr}_{\mathit{loadHtsSt}}, \mathit{Adr}_{\mathit{deloadHtsSt}})\big)\big) \\ \qquad\qquad\qquad \backslash \mathit{PrivHts} \end{array}$$

with the following sets of events used for synchronisation and hiding:

$$\begin{aligned} A_{\mathit{Driver}} &= \{\mathit{arrived}, \mathit{drive}, \mathit{position}\} \\ A_{\mathit{Acqui}} &= \{\mathit{acquireOrder}, \mathit{newOrder}, \mathit{listen}, \mathit{offer}, \mathit{order}\} \\ A_{\mathit{Ctrl}} &= \{\mathit{loadHtsWzm}, \mathit{deloadHtsWzm}, \mathit{loadHtsSt}, \mathit{deloadHtsSt}, \\ &\qquad \mathit{newOrder}, \mathit{acquireOrder}, \mathit{arrived}, \mathit{drive}\} \\ \mathit{PrivHts} &= \{\mathit{arrived}, \mathit{drive}, \mathit{position}, \mathit{acquireOrder}, \mathit{newOrder}\} \end{aligned}$$

For reasons of readability we have unfolded the iterated alphabetised parallel composition in the process equation for *Hts*. The renaming P_{CC} is in this case empty since we have adopted the convention that omitted connector names equal their port names.
Next, the process equation for the multi-capsule *MHts* is constructed:

$$\begin{array}{l} \mathit{MHts}(\mathit{Adr}_{\mathit{loadHtsWzm}}, \mathit{Adr}_{\mathit{deloadHtsWzm}}, \\ \qquad \mathit{Adr}_{\mathit{loadHtsSt}}, \mathit{Adr}_{\mathit{deloadHtsSt}}, \mathit{Adr}_{\mathit{offer}}) = \\ \quad |||_{in:\{T1,T2\}} \bullet \mathit{Hts}(\mathit{Adr}_{\mathit{loadHtsWzm}}, \mathit{Adr}_{\mathit{deloadHtsWzm}}, \\ \qquad\qquad \mathit{Adr}_{\mathit{loadHtsSt}}, \mathit{Adr}_{\mathit{deloadHtsSt}}, \mathit{Adr}_{\mathit{offer}})[\![in]\!] \end{array}$$

Finally we can give the system description, applying once again the translation scheme for compound capsules:

$$\begin{array}{l} \mathit{System} = \\ \quad \big(\mathit{MHts}(\{A,B,C\},\{A,B,C\},\{\mathit{In},\mathit{Out}\},\{\mathit{In},\mathit{Out}\},\{A,B,C\}) \\ \qquad\qquad {}_{A_{\mathit{MHts}}}||_{A_{\mathit{MWzm}} \cup A_{\mathit{MStore}}} \\ \qquad \big(\mathit{MWzm}(\{T1,T2\},\{T1,T2\})\ {}_{A_{\mathit{MWzm}}}||_{A_{\mathit{MStore}}}\ \mathit{MStore}\big)\big) \setminus \mathit{PrivSys} \end{array}$$

with synchronisation sets and private channels

$$A_{\mathit{MHts}} = \{\mathit{loadHtsWzm}, \mathit{deloadHtsWzm}, \mathit{loadHtsSt}, \mathit{deloadHtsSt},$$

$$\begin{aligned}
&\qquad\qquad \mathit{listen}, \mathit{offer}, \mathit{order}\}\\
A_{MWzm} &= \{\mathit{loadHtsWzm}, \mathit{deloadHtsWzm}, \mathit{offer}, \mathit{order}, \mathit{request}\}\\
A_{MStore} &= \{\mathit{loadHtsSt}, \mathit{deloadHtsSt}\}\\
\mathit{PrivSys} &= A_{MHts} \cup A_{MWzm} \cup A_{MStore}
\end{aligned}$$

This completes the translation.

5 Conclusion

In this paper, we have proposed a translation of UML-RT structure diagrams into CSP. This allows us to use graphical architecture descriptions in CSP-OZ without losing its formal semantics. The technique is not only applicable to CSP-OZ specifications, but more generally to all specification languages which use CSP for structure descriptions. The only change needed then is the interpretation of basic capsules, which stand for CSP-OZ classes in our case but may also be interpreted differently. The translation gives *one* possible semantics to UML-RT collaboration diagrams. In a different setting (for instance hardware design on a much lower abstraction level), a different semantics might be conceivable.

The basis for the translation given in this paper is a formalisation of the syntax of UML-RT structure diagrams in Object-Z. This is similar to the work done in [KC99], which formalises the syntax of UML class diagrams with Z and uses the formalisation for a translation of class diagrams to Object-Z classes.

So far we have not explicitly treated *protocols*, which are also part of UML-RT. Protocols are used for specifying the type of interactions which may take place over some connector. A protocol can for instance define a set of signals passed over a connector, or can give a valid communication sequence. Since the basic communication paradigm of CSP is *synchronous* communication, we have assumed that all protocols define synchronous communication over a single channel. However, we envisage the possibility of using more elaborate protocols in an architecture description, for instance protocols for defining asynchronous communication or communication over unreliable channels. To fit into the CSP view on UML-RT diagrams, these protocols should be specified in CSP. This approach to protocol definition is similar to the method chosen in WRIGHT [AG97], an architecture description language (ADL) based on CSP. In WRIGHT, an architecture description consists of a set of components together with a collection of *connectors* which are given in CSP.

Another work similar to ours is the ADL Darwin [MDEK95]. Darwin is both a graphical and a textual modelling language; it has a formal semantics in terms of Milner's π-calculus [Mil99]. The usage of a calculus with mobility is necessary there because Darwin allows to specify dynamically evolving system structures. Since our goal was to find a graphical description of *CSP* structure specifications, we had no need for incorporating facilities for describing mobility.

A completely different semantic approach to UML-RT diagrams can be found in [GBSS99]. There the focus is on using a visual but still well-defined formalism (*interaction graphs*) for interpreting UML-RT diagrams.

References

[AG97] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 1997.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language user guide*. Addison Wesley, 1999.

[CE98] T. Clark and A. Evans. Foundations of the unified modeling language. In *Northern Formal Methods Workshop*, Electronic Workshops in Computing. Springer, 1998.

[DLC98] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. Translating the OMT dynamic model into Object-Z. In J.P. Bowen, A. Fett, and M.G. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*, volume 1493 of *Lecture Notes in Computer Science*, pages 347–366. Springer, 1998.

[Fis97] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, volume 2, pages 423–438. Chapman & Hall, 1997.

[Fis00] C. Fischer. *Combination and Implementation of Processes and Data: From CSP-OZ to Java.* PhD thesis, Bericht Nr. 2/2000, University of Oldenburg, April 2000.

[FR99] R. France and B. Rumpe, editors. *UML'99: The Modified Modeling Language – Beyond the Standard*, volume 1723 of *Lecture Notes in Computer Science.* Springer, 1999.

[GBSS99] R. Grosu, M. Broy, B. Selic, and G. Stefanescu. What is behind UML-RT? In *Behavioural Specifications of business and systems.* Kluwer, 1999.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[KC99] S.-K. Kim and D. Carrington. Formalizing the UML class diagram using Object-Z. In R. France and B. Rumpe, editors, *UML'99: The Unified Modelling Language – Beyond the Standard*, volume 1723 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 1999.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *ESEC '95: European Software Engineering Conference*, 1995.

[Mil99] R. Milner. *Communicating and Mobile Systems: The π-Calculus.* Cambridge University Press, 1999.

[Obj99] Object Management Group. *OMG Unified Modeling Language Specification*, June 1999. version 1.3.

[OR00] E.-R. Olderog and A.P. Ravn. Documenting design refinement. In M.P.E. Heimdahl, editor, *Proc. of the Third Workshop on Formal Methods in Software Practice*, pages 89–100. ACM, 2000.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1997.

[Smi00] G. Smith. *The Object-Z Specification Language.* Kluwer Academic Publisher, 2000.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall International Series in Computer Science, 2nd edition, 1992.

[SR98] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. Technical report, ObjecTime, 1998.

[Weh00] H. Wehrheim. Specification of an automatic manufacturing system – a case study in using integrated formal methods. In T. Maibaum, editor, *FASE 2000: Fundamental Aspects of Software Engineering*, number 1783 in LNCS. Springer, 2000.