Interaction Interfaces – Towards a scientific foundation of a methodological usage of Message Sequence Charts

Manfred Broy and Ingolf Krüger
Technische Universität München
Arcisstr. 21
D-80290 München
Germany

E-mail: {broy|kruegeri}@in.tum.de

Abstract

We introduce the formal notion of an interaction interface. *Its purpose is to specify formally the interaction between two or more components that co-operate as subsystems of a distributed system. We suggest the use of interaction interfaces for the description not of the behaviour of a single component in isolation but of the interface, the co-operation, between two or more components that are interacting within a distributed system. Typical examples are the interaction between an embedded system and its environment or the interaction between a sender and a receiver in a communication protocol. An interaction interface can be formally described by predicates characterising sets of interaction histories. We understand the specification of interaction histories as a typical step in system development that prepares the decomposition of a system into interacting subcomponents. After fixing the distribution structure of the system, an interaction interface is worked out that describes how the introduced subcomponents interact. In a successive development step we systematically derive the individual component specifications from the interface description. We show how such an interaction interface can be decomposed systematically into component specifications.*

1. Introduction

When developing distributed and interacting systems, a classical top down proceeding begins with a *problem analysis* followed by a *requirements capture* and the *specification* of the required system services. The specification describes the required black box behaviour of the overall system. In further development steps, the system is decomposed into subcomponents forming the constituents of the system architecture.

In such a systematic development process, we are, in a first step, not mainly interested in the specification of the behaviour of the various components of a system, called the *component black box specification*, but rather in the description of the interaction between these components. Such a proceeding is typical for the development steps of software architectures where we are, in particular, aiming at the decomposition of a system into subsystems, called the components of the system, that realise the required system services by means of co-operation. We call the description of the interaction between components an *interaction interface specification*. After fixing the architecture, we develop component specifications based on the description of the interaction.

There are several branches of software engineering that suggest the modelling of the interaction between components as a major step in requirements engineering or in design, preparing and complementing the modelling and specification of the behaviour of components in isolation. This is typical for the development of reactive embedded systems, for software architectures, for a number of design patterns (see [6]), and for the description of communication protocols in telecommunication applications. There, mainly graphical descriptions of interaction instances are used such as the *message sequence charts* (MSCs) as they are described in [9] as a spin-off of SDL or the *process diagrams* of GRAPES (see [7]). MSCs have found their way also into object-oriented analysis techniques for representing so-called *use-cases* (see [8]) or scenarios (cf. [10, 2, 5]). We consider MSCs as an example of a semiformal description technique that can be turned into a formal description method by providing a formal semantics. We do not present

such a formal definition of the meaning of MSCs here, but work with a formal notion of instances of interaction right away.

In the following, we show how the notion of an interaction interface can be described by formal means and how an interaction interface can be systematically refined into component descriptions. We introduce, in particular, a formal model for interaction interfaces and relate it to a formal model of the functional behaviour of components.

The paper is structured as follows: in Section 2 we introduce the formal concept of an interaction interface. Section 3 provides an analysis of the idea of an interaction interface and introduces general timing and causality requirements for interface specifications. Furthermore, other techniques for modelling interaction interfaces such as traces and graphical models are briefly discussed. In Section 4 we show how to derive component specifications from an interaction interface specification. In Section 5 we discuss differences between safety and liveness properties in the deduction of a component specification from an interface specification. In Section 6 we study the concept of refinement for interaction interface specifications. In Section 7 we relate this discussion to the notion of realisability of components. Section 8 contains our conclusions.

2. Interaction interfaces

An interaction interface describes the mutual interaction between two or more components. We concentrate, for simplicity and convenience, in the following mainly on the interaction interface between two components. However, all the ideas presented for this restricted case easily generalise and carry over to the description of the interaction of larger sets of components. Moreover, also the interaction between a system and its environment can be seen as a special case of interaction interfaces between two components. We work with a very simple, yet powerful model

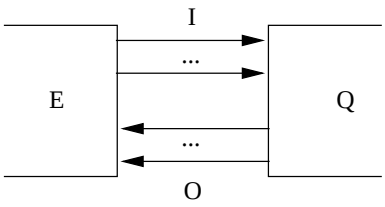

Figure 1. Graphical Description of the Interaction Interface Between Two Components

of interaction. We assume that the components of a system communicate by exchanging messages asynchronously via directed channels. An interaction interface between two components, as it is schematically illustrated in Fig. 1, consists of a *syntactic* and of a *semantic part*. The syntactic part is rather straightforward. It is specified by the names, sorts, and directions of the channels that connect the two components. In this document we graphically denote a component by a box labeled with the component's name. We represent channels by arrows directed from the source to the destination component. Let C denote the set of all channels that are part of the interaction interface. We assume that the set of channels C can be decomposed into the set I of channels directed from left to right in Fig. 1 and the set O of channels directed from right to left. For concrete applications we assume, of course, that the channels are typed. More precisely, each channel is assigned a data type that indicates the sort of the messages that are sent over this channel.

It is helpful to introduce some basic notions and some specific notation before we go deeper into our subject. We use *streams*, i.e. finite or infinite sequences of messages, to represent communication histories. A *timed stream* (more precisely a stream with discrete time) of messages from the set M is an element of the set

$$(M^*)^\infty$$

which stands for

$$\mathbb{N}\backslash\{0\} \to M^*.$$

A timed stream consists of an infinite sequence of finite sequences of messages. The k-th of the finite sequences in such a stream represents the sequence of messages transmitted on the channel in the k-th time interval. Given $s \in (M^*)^\infty$ and $k \in \mathbb{N}$ by $s{\downarrow}k$ we denote the stream of the first k time intervals. Formally $s{\downarrow}k$ is the restriction of the domain of $s : \mathbb{N}\backslash\{0\} \to M^*$ to the finite set $\{1, \ldots, k\}$ with

$$(s{\downarrow}k).j = s.j \text{ for } 1 \leq j \leq k.$$

By $<m_1\ m_2\ \ldots\ m_n>$ we denote the finite stream consisting, in that order, of the elements m_1 through m_n with $m_i \in M$ for $1 \leq i \leq n$. We denote the empty stream by $<>$. Given two (finite or infinite) streams s_1 and s_2, we denote their concatenation by $s_1{\hat{}}s_2$. The operation $N\#s$ yields the number of messages from set N in stream s. If s is a timed stream and $t \in \mathbb{N}\backslash\{0\}$, we write $s : t$ for the finite sequence of messages occuring in s at time point t. For any set N we denote the number of its elements by $\#N$.

Given a set of channels C we write $\vec{C}$ for the set of all its *valuations*. Valuations are the mappings

$$C \to (M^*)^\infty.$$

The elements of the set $\vec{C}$ are also called the *communication histories* for the channels in C. Every valuation $x \in \vec{C}$ of the channels in C by timed streams assigns a communication history $x.c$ to every channel $c \in C$. In such a history,

we represent the stream of the sequences of messages sent on the channels in each of the time intervals. If the channels are typed then we assume for a valuation $x \in \vec{C}$ that the stream $x.c$ for every channel $c \in C$ contains only messages of the respective sort.

Given a valuation $x \in \vec{C}$ and a number $k \in \mathbb{N}$, by

$$x{\downarrow}k$$

we define the finite valuation in

$$C \to (M^*)^*$$

by

$$((x{\downarrow}k).c) = (x.c){\downarrow}k.$$

Thus $x{\downarrow}k$ denotes the finite initial segment of the messages of the first k intervals.

Given a valuation $x \in \vec{C}$ we denote by $x|_I$ the restriction of the valuation x to the channels in the set $I \subseteq C$; $x|_I$ is a valuation in the set $\vec{I}$, specified by the formula

$$(x|_I).c = x.c \;\Leftarrow\; c \in I.$$

Throughout this paper, we assume that the set C of channels is decomposed into channels in the set I pointing into one direction and channels in the set O pointing into the opposite direction. Mathematically speaking, we assume $C = I \cup O$ and $I \cap O = \emptyset$.

Given channel valuations $i \in \vec{I}$ and $o \in \vec{O}$ we write $i \oplus o$ for the valuation in $\vec{C}$ where for all channels $c \in C$ we specify this valuation by the formulas

$$(i \oplus o).c = i.c \;\Leftarrow\; c \in I$$

$$(i \oplus o).c = o.c \;\Leftarrow\; c \in O$$

Given these basic definitions the interaction interface can be specified mathematically by a predicate R that describes the set of valuations for the channels in C by streams that may occur during the interaction between the two components. Formally, R corresponds to a mapping

$$R : \vec{C} \to \mathbb{B}$$

Note that, of course, both components may have further input and output channels that are not part of the interaction under consideration. In the interface description these channels may be ignored as long as we do not want to describe situations where the interaction interface depends essentially on the valuations of these additional channels[1]. Only then this dependency has to be made explicit in the interaction interface. In the following, we ignore the latter case. Then the interaction interface describes all communication histories allowed for the channels in C that may occur for the set of possible evaluations for all channels (including, in particular, the channels not contained in C).

[1] In the literature, authors speak of hidden channels.

3. An operational view on interaction interfaces

In the model introduced above, where we describe the interaction interface by a predicate $R : \vec{C} \to \mathbb{B}$ we assumed a partitioning of the channel set C into the sets I and O, but we did not take into account the directedness of these channels, semantically. Intuitively, it is clear that the messages on the channels in I in Fig. 1 are sent and, therefore, exclusively determined by the component on the left while the messages on the channels in O are determined exclusively by the component on the right. This causal relationship between the messages and their timing is discussed in more detail in this section.

We think about the interaction of the two components in Fig. 1 as follows. In every time interval both components determine independently their sequences of messages that they send on their outgoing channels in that time interval. This idea captures the essence of asynchronous input and output in directed communication and leads to the formulation of the following property that we require to hold for any proper specification of a semantic interaction interface.

The semantic interface described by the predicate R is called *consistent with respect to time guardedness* if for all valuations $i, i' \in \vec{I}$ of the input channels we have for all times $t \in \mathbb{N}$:

$$\begin{array}{c} i{\downarrow}t = i'{\downarrow}t \\ \Rightarrow \{o{\downarrow}t{+}1 : R(i \oplus o)\} = \{o{\downarrow}t{+}1 : R(i' \oplus o)\} \end{array}$$

as well as for all valuations $o, o' \in \vec{O}$ of the output channels we have for all times $t \in \mathbb{N}$:

$$\begin{array}{c} o{\downarrow}t = o'{\downarrow}t \\ \Rightarrow \{i{\downarrow}t{+}1 : R(i \oplus o)\} = \{i{\downarrow}t{+}1 : R(i \oplus o')\} \end{array}$$

Time guardedness expresses that a component as well as the environment may react to input received in the t-th time interval only in the $(t{+}1)$-th time interval[2] or in later time intervals. Since we assume that the only way of interaction between a component and its environment is by sending messages on the channels[3], both the output produced by the component and the environment in the $(t{+}1)$-th time interval must only depend on the input they have received until time interval t from their partner component. If every reaction to an incoming message requires some time and, therefore, is slightly delayed, time guardedness is a reasonable assumption[4].

[2] This is a reasonable assumption as long as we assume that our time granularity is chosen so fine that reaction to input can always only occur as early as in the next time interval.

[3] This excludes a connection by hidden channels that provide information about the future messages for the visible channels.

[4] There are other models of interactive systems, of course, that do not follow such a strict idea of causality.

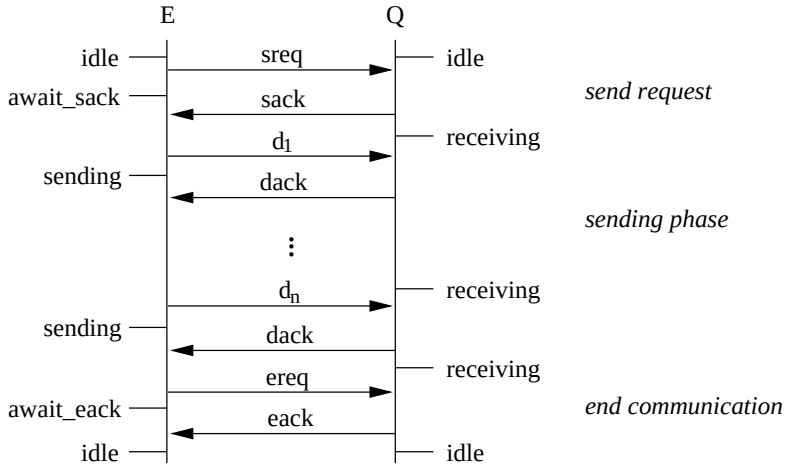

Figure 2. Message Sequence Chart for the Standard Case of Successful Communication, with Component E Initiating the Transmission

Time guardedness is a simple idea to model and ensure the principle of *causality* between the messages mutually exchanged in an interaction interface. If the messages are fixed until time t, the messages on the channels in I as well as the messages on the channels in O can be and have to be chosen independently of each other in the $(t+1)$-th time interval. More precisely, which messages occur in the $(t+1)$-th interval on the channels in the set I may depend on the messages on all the channels in $I \cup O$ until time t but not on the messages in the $(t+1)$-th interval on the channels in the set O and vice versa. This concept is to be seen in contrast to the ideas of instantaneous reaction and *perfect synchrony* (see Esterel in [1]) where the time model is chosen in a way that supports a feedback process taking place between two components within a single time slot.

Another possibility, apart from valuations of directed channels by streams, for modelling interaction interfaces are sets of traces. A trace can be associated with a time guarded function f easily as follows. Seen from the perspective of the component to the right in Fig. 1, given the input sequences $i_k : I \to M^*$ for the k-th time interval we get the input stream

$$<i_1\ i_2\ i_3\ i_4\ \ldots>$$

and given the output sequences $o_k : O \to M^*$ we get the output stream

$$<o_1\ o_2\ o_3\ \ldots>.$$

We construct from these two streams the trace consisting always of the output at time point k followed by the input at time point k[5]. The input and output at time point k are separated from the input and output at time point $k+1$ by a time tick signal $\surd$. In addition, we may label these messages by the channel names and interleave these labelled messages in the sequences i_k and o_k leading to the sequences $\tilde{o}_k$ and $\tilde{i}_k$. We obtain traces of the form

$$\tilde{o}_1 \hat{}\, \tilde{i}_1 \hat{}\, <\surd> \hat{}\, \tilde{o}_2 \hat{}\, \tilde{i}_2 \hat{}\, <\surd> \hat{}\, \ldots$$

In traces, in contrast to our interaction predicates, concurrency is eliminated and the behaviour is represented by linear sequences with the help of interleaving.

More common, in practice, than logical techniques are graphical descriptions of interaction interfaces such as MSCs (see [9]) that – in the special case of describing the interaction between two components – are a graphical representation of traces. An example is shown in Fig. 2. Here, the labels to the left of E's axis, and to the right of Q's axis denote E's and Q's state when sending or receiving messages, respectively. The italic labels to the right of the figure describe the phase of the communication protocol. However, in contrast to MSCs that describe only one finite instance of a case of interaction (also called a use case or scenario), interaction interfaces represented by predicates characterise *all* the observable interactions and, therefore, give a comprehensive description of component requirements.

Example 1 (A simple two-way protocol) We study a simple, symmetric interaction protocol that supports the interaction between two components very much along the lines of the ABRACADABRA-protocol (see [3]). In the ideal case the interaction proceeds as shown in Fig. 2. In fact, the idea of the protocol is that each of the components, as long

[5] Note that the choice of the order between the messages from o_k and i_k in the trace is arbitrary, since due to the causal independence of these messages in each time interval they can be represented in any order.

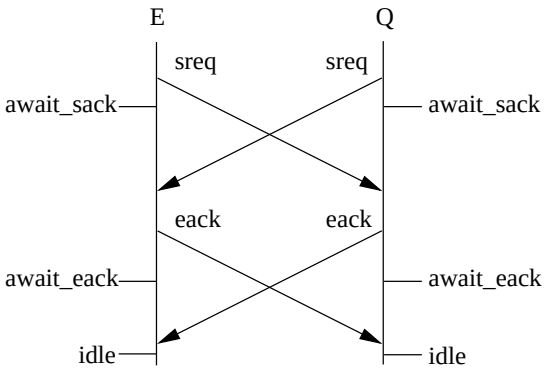

Figure 3. Message Sequence Chart for Conflicting Interactions

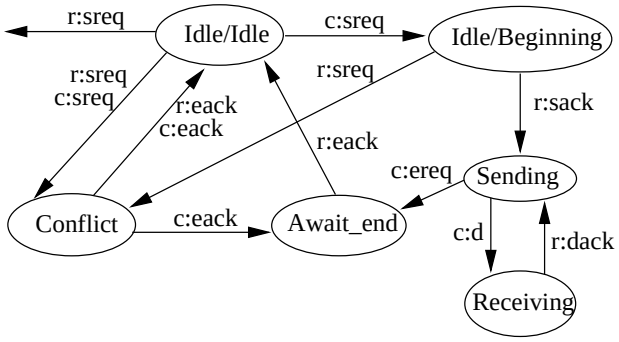

Figure 4. State Transition Diagram for the System with two Components.

as it is idle, is free to choose to become a sender and to start a transmission activity with the other component that should act as a receiver. A problem may occur in this protocol if both components begin to act as a sender simultaneously and so try to start their communication concurrently. An idea to deal with this situation is that in such a case both components terminate their transmission and retry later. An MSC that describes this case is shown in Fig. 3.

From these two cases of interaction (the two use cases) we may conclude the following requirements of the components. The first message a component may send or receive is the message sreq. If a component received the message sreq it must send the message eack, if it did not send the message sreq itself already before, otherwise it has to send the message ereq. In the second case the transmission is finished and the component gets into its initial state again. Otherwise the communication proceeds by the component receiving a data element and sending the acknowledgement message dack in return. This ping-pong goes on until the message ereq issued by the sender indicates the end of the transmission.

Of course, this is a very simple case where the two use cases do not overlap as this might happen for components that may carry out several independent communications by multiplexing. By the way, the question whether several use cases described by MSCs should be carried out interleaved or not is often not answered explicitly in the description methods. In principle, it is necessary to add textual comments that indicate in which way the MSCs should be interpreted. What is going on in the protocol can also be specified by a state transition diagram for the states of both components as it is given in Fig. 4. We give only one half of it since the other half is symmetric. Fig. 5 gives the state transition diagram for the individual component. Here the transitions are labelled by patterns of the form m_1/m_2 where m_1 is the input message that triggers the transition and m_2 is the output generated in this transition. If the messages m_1 or m_2 are replaced by "-" this means that no input is required to trigger the transition or no output message occurs during the transition, respectively. A formal specification of

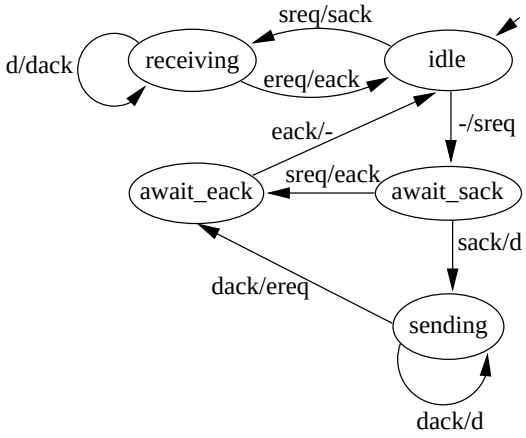

Figure 5. State Transition Diagram of the Communication Protocol

this interaction interface is given by a predicate

$$R : (\{c, r\} \to (M^*)^\infty) \to \mathbb{B}$$

where c and r are the channels that connect the components. Here

$$M = \{sreq, sack, dack, ereq, eack\} \cup D$$

where D is a set of data messages. To describe the interaction interface formally we first specify a family of simple predicates $R_t : (\{c, r\} \to (M^*)^\infty) \to \mathbb{B}$ by

$$R_t.x \equiv G(x.c, x.r, t) \wedge G(x.r, x.c, t)$$

where the auxiliary predicate G is specified by the following equation

$$\begin{aligned} G(s,v,t) \quad &\equiv \quad idle(s,v,t) \\ &\vee \quad receiving(s,v,t) \\ &\vee \quad sending(s,v,t) \\ &\vee \quad await\_sack(s,v,t) \\ &\vee \quad await\_eack(s,v,t) \end{aligned}$$

Each predicate like $idle$, $receiving$, $sending$, etc., specifies the history of inputs and outputs until time point t that corresponds to the respective state of the automaton from Fig. 5. We do not give the specifications for all state predicates explicitly, since they can be schematically derived from the state transition diagram as follows. To determine the state predicate $\sigma(s,v,t{+}1)$ that corresponds to state σ, we consider all transitions whose target state is σ. For each such transition we add the disjunctive clause

$$(expect(\sigma', s, v, t{+}1, m) \wedge (s : t{+}1 = {<}m'{>}))$$

to the definition of $\sigma(s,v,t{+}1)$, if the transition starts in state σ' and is labeled with m/m'. Intuitively, $expect(\sigma', s, v, t, m)$ yields true if and only if the expected input m has been received in state σ' strictly before time t. More precisely, we define

$$\begin{aligned} expect(\sigma', s, v, 0, m) \quad &\equiv \quad \text{false} \\ expect(\sigma', s, v, t{+}1, m) \quad &\equiv \quad (\sigma'(s,v,t) \wedge v : t = {<}m{>}) \\ &\vee \quad (\quad expect(\sigma', s, v, t, m) \\ &\qquad \wedge \ s : t = {<>} \\ &\qquad \wedge \ v : t = {<>}) \end{aligned}$$

Furthermore, we add the disjunctive clause

$$(\sigma(s,v,t) \wedge s : t{+}1 = {<>} \wedge v : t{+}1 = {<>})$$

which ensures that a component may remain in its state if it neither receives input, nor produces output at time $(t{+}1)$. If σ is an initial state, we also define $\sigma(s,v,0) \equiv$ true, and $\sigma(s,v,0) \equiv$ false otherwise. This way the transition diagram is translated into a set of inductively defined predicates. As an example consider predicate $idle$, which is defined as follows:

$$\begin{aligned} idle(s,v,0) \quad &\equiv \quad \text{true} \\ idle(s,v,t{+}1) \quad &\equiv \quad (expect(receiving, s, v, t{+}1, ereq) \\ &\qquad \wedge \ s : t{+}1 = {<}eack{>}) \\ &\vee \quad (expect(await\_eack, s, v, t{+}1, eack) \\ &\qquad \wedge \ s : t{+}1 = {<>}) \\ &\vee \quad (idle(s,v,t) \\ &\qquad \wedge \ s : t{+}1 = {<>} \wedge v : t{+}1 = {<>}) \end{aligned}$$

The other predicates are specified in the same style. For a state predicate like $receiving$, we can use existential quantification to denote that in each transition labeled with d a different input value for d may occur.

So far we did not consider liveness properties. For the protocol discussed above the latter are very simple since each message requires exactly one reply. Moreover, if there is a conflict, both components have to issue a message in return eventually. These two properties are subsumed by the following definition:

$$\begin{aligned} R.x \quad &\equiv \quad (\forall t \in \mathbb{N} : R_t.x) \\ &\wedge \quad \#\{t \in \mathbb{N} : idle(x.c, x.r, t) \wedge idle(x.r, x.c, t)\} \\ &\qquad = \infty \end{aligned}$$

This gives the formal specification of the interaction interface R. The state transition diagram in Fig. 5 can be generated fully automatically from the MSCs. □

We will not come back to the question of the automatic generation of state transition diagrams from MSCs in the course of this paper. This is an issue we plan to investigate in a forthcoming paper.

4. From interaction interfaces to component specifications

In this section we introduce the concept of a system component and show how to derive component specifications in a systematic way from interaction specifications. A relational component specification is given by a predicate

$$F : \vec{I} \to (\vec{O} \to \mathbb{B})$$

that fulfils a number of simple properties with respect to time flow. For a component specification we require the following properties:

1. F is *time guarded*. Time guardedness of F is formally defined as follows:

$$\begin{gathered} i{\downarrow}t = i'{\downarrow}t \\ \Rightarrow \{o{\downarrow}t{+}1 : F(i).o\} = \{o{\downarrow}t{+}1 : F(i').o\} \end{gathered}$$

2. F is *realisable*. Realisability of the function F is formally defined as follows:

$$[\![F]\!] \neq \emptyset$$

By $[\![F]\!]$ we denote the set of time guarded functions

$$f : \vec{I} \to \vec{O}$$

that are contained pointwise in the function F; then, formally, $(F.i)(f.i)$ holds for all input histories $i \in \vec{I}$. The functions f are timed guarded, if:

$$i{\downarrow}t = i'{\downarrow}t \Rightarrow (f.i){\downarrow}t{+}1 = (f.i'){\downarrow}t{+}1$$

F is called *fully realisable* if for all input histories $i \in \vec{I}$ we have

$$\{o \in \vec{O} : F(i).o\} = \{f.i \in \vec{O} : f \in [\![F]\!]\}$$

Only if a component is realisable we can give an implementation without any assumptions about the environment in which it is used. The critical and in the course of the development decisive step in the decomposition of a system into subsystems is, after the interaction interface has been specified, the transition from an interaction interface specification (where again $C = I \cup O, I \cap O = \emptyset$)

$$R : \vec{C} \to \mathbb{B}$$

to a component specification

$$Q : \vec{I} \to (\vec{O} \to \mathbb{B})$$

for the components involved. We assume for simplicity that the interface describes the interaction between only two components called E and Q as illustrated in Fig. 1.

Again, the set of channels C connecting the components E and Q is partitioned into input channels I and output channels O for the component Q. This leads to component specifications (for simplicity we identify the component names with the names of the specifying predicates)

$$Q : \vec{I} \to (\vec{O} \to \mathbb{B})$$

$$E : \vec{O} \to (\vec{I} \to \mathbb{B})$$

The interaction interface specification R defines the set of possible communication histories for the channels between the components E and Q. From R we deduce the requirements (specifying formulas) for the specifications E and Q. We demonstrate how to do this only for component Q. For component E this can be done in a fully symmetric way.

Technically, we show how to decompose the predicate R describing the interaction between the two components into two predicates for the specification of Q, called the *assumption* A and the *commitment* G. The idea behind this decomposition can be explained as follows.

The communication histories $x \in \vec{C}$ for the channels in the interaction interface are divided by a restriction of x onto the channels in the sets I and O into an input history $x|_I$ and an output history $x|_O$, respectively. In general, not for every valuation $i \in \vec{I}$ there exists a communication history x with $R.x$ such that $i = x|_I$. The predicate G (for guarantee) is used to characterise the set of output histories $o \in \vec{O}$ that are admitted for the input histories i, which fulfil the assumption A.

However, even for input streams $i \in \vec{I}$ for the component Q that do not fulfil the assumption (in other words: there does not exist a communication history $x \in \vec{C}$ such that $R.x$ holds with $x|_I = i$) the output of the component Q for the input history i may be restricted by the specification. If there exists a valuation $x \in \vec{C}$ such that $R.x$ holds and some time point $k \in \mathbb{N}$ such that $x|_I \downarrow k = i \downarrow k$, the output of Q for the input history i must fulfil the requirements expressed by predicate R at least until time point k according to our requirement for time guardedness.

We work with the following predicates (for $t \in \mathbb{N}$)

$$A_t : \vec{I} \times \vec{O} \to \mathbb{B}$$

$$A' : \vec{I} \to \mathbb{B}$$

that we use to construct the assumption predicate A for the component Q based on the interaction interface specification R. We specify them by the following equations (for all times $t \in \mathbb{N}$):

$$\begin{aligned} A_t(i,o) &\equiv \exists i' \in \vec{I}, o' \in \vec{O} : \\ &\quad i' \downarrow t = i \downarrow t \wedge o' \downarrow t = o \downarrow t \wedge R(i' \oplus o') \\ A'(i) &\equiv \exists o : R(i \oplus o) \end{aligned}$$

The predicate A_t characterises those input histories whose finite prefixes until time point t may actually occur on the channels in I in the interaction between the components E and Q. The assumption predicate A_t defines the safety properties that can be assumed for the environment according to the interaction as described in the interaction interface. The assumption A' formalises the requirement that the complete input history i does occur in an interaction. The requirements that the component Q is supposed to fulfil provided the assumption holds are formalised with the help of the auxiliary predicates (for $t \in \mathbb{N}$)

$$G_t : \vec{I} \times \vec{O} \to \mathbb{B}$$

which we use to formulate the commitment. The predicate G_t is specified as follows

$$\begin{aligned} G_t(i,o) &\equiv \exists i', o' : \\ &\quad i' \downarrow t = i \downarrow t \ \wedge \ o \downarrow t+1 = o' \downarrow t+1 \\ &\quad \wedge \ R(i' \oplus o') \end{aligned}$$

Note the asymmetry in this definition in the treatment of i and o. This is motivated by the requirement of time guardedness. We specify the component Q with the help of the predicates A_t, A', and G_t by the following formula:

$$\begin{aligned} Q(i).o &\equiv (\forall t \in \mathbb{N} : A_t(i,o) \Rightarrow G_t(i,o)) \\ &\wedge \ (A'(i) \Rightarrow R(i \oplus o)) \end{aligned}$$

The first part of the formula defines the safety properties for the component Q. The second part defines the liveness properties for Q. The second clause may be weakened by dropping those liveness assumptions the partner component

is supposed to take care of. We will come back to this point in Section 5.

Note that the predicate A' subsumes all the predicates A_t since the proposition

$$A'(i) \Rightarrow \exists o \in \vec{O} : \forall t \in \mathbb{N} : A_t(i,o)$$

holds. This is a straightforward consequence of the definitions of A' and A_t. However, the reverse direction does not hold, in general.

Note, moreover, that if the predicate R is consistent with respect to time guardedness we have

$$\forall t \in \mathbb{N}, i \in \vec{I} :$$
$$A_t(i,o) \Rightarrow \exists o' \in \vec{O} : o{\downarrow}t{+}1 = o'{\downarrow}t{+}1 \wedge G_{t+1}(i,o')$$

This is easily proved by induction on t. This shows that the pure safety condition

$$Q'(i).o \equiv \forall t \in \mathbb{N} : A_t(i,o) \Rightarrow G_t(i,o)$$

fulfils the property

$$\forall i \in \vec{I} : \exists o \in \vec{O} : Q'(i,o) \qquad (1)$$

that indicates that the specification Q' is consistent in the sense that for every input history there exists an output history. We prove formula (1) by contradiction. Assume that formula (1) is false. Then there exists some input history $i \in \vec{I}$ and some time point $t \in \mathbb{N}$ such that for all output histories $o \in \vec{O}$ we have:

$$A_t(i,o) \wedge \neg G_t(i,o)$$

This is in contradiction to the time guardedness of R since $A_t(i,o)$ implies that there exists an input history i', and an output history o' such that the following formula holds:

$$i'{\downarrow}t = i{\downarrow}t \wedge o'{\downarrow}t = o{\downarrow}t \wedge R(i' \oplus o')$$

By time guardedness we obtain that there also exists an output history o'' such that

$$i'{\downarrow}t = i{\downarrow}t \wedge o''{\downarrow}t{+}1 = o{\downarrow}t{+}1 \wedge R(i' \oplus o'').$$

This proves the validity of the formula

$$A_t(i,o) \wedge G_t(i,o),$$

which is a contradiction to the assumption that formula (1) is false.

In fact, the step from an interaction interface specification to a component specification can be done in a rather systematic, canonical way due to the required property of time guardedness.

As pointed out in Section 3, the computational model associated with an interaction interface is rather straightforward. A chain (i_t, o_t) of valuations by finite streams consisting of pairs of histories until the time point t is generated. In the $(t{+}1)$-th time interval the output of the components on their output channels can be determined taking into account only the history (i_t, o_t)[6]. The transition from an interaction interface to a component specification works nicely for safety properties. In fact, these can be uniquely determined for the components, but this does not hold for the liveness properties. To make the methodological difference between safety and liveness properties precise we give the following formal characterisation of safety and liveness. A predicate $P(x,y)$ is called a *safety relation* for a component Q with input histories $x \in \vec{I}$ and output histories $y \in \vec{O}$ if we have the validity of the following equivalence

$$P(x,y) \quad \equiv \quad \forall t \in \mathbb{N} : \exists y' \in \vec{O} : y{\downarrow}t = y'{\downarrow}t \wedge P(x,y')$$

P is called a *liveness relation* for the component Q if (for all histories $x \in \vec{I}$ and $y \in \vec{O}$)

$$\forall t \in \mathbb{N} : \exists y' \in \vec{O} : y{\downarrow}t = y'{\downarrow}t \wedge P(x,y')$$

Note the asymmetry between the usage of histories x and y in each of these formulas, which reflects the directedness of input and output for the component Q. A detailed discussion of the motivation for these differences is found in [4].

5. Decomposing liveness

The crucial difference between a liveness and a safety condition in an interaction interface can be explained as follows: for a communication history that violates a safety condition we can always find a minimal time interval in which the condition is violated. This way it is always possible to determine precisely which of the components violated the safety conditions[7].

For liveness conditions the situation is more subtle. If a liveness condition that is part of an interaction interface is violated then we cannot always uniquely identify which of the components is responsible for that violation. This makes it more difficult to decide how to handle liveness conditions when deriving component specifications from interface specifications. In contrast, safety conditions are uniquely determined for each component by a given interface specification.

In the transition from an interaction interface to an individual component specification the responsibility for safety

[6] Note that time guardedness is exactly what guarantees the possibility of the independence of the choices of the output messages by the component and by its environment.

[7] In extreme cases, both components violate the safety conditions independently. Then both behave incorrectly.

properties is obvious. In the $(t+1)$-th interval the possible messages on the channels in I determine the assumption for the next step and the possible messages on the channels in O determine the commitment of the current step. Therefore, for a given interaction interface specification all safety properties can uniquely be decomposed into safety properties for the component Q and for its environment E.

For the liveness properties this is, in principle, also true. If we have to be prepared to deal with any possible behaviour included in the interaction interface for the other component then we have to choose the liveness property as strong as possible. It may be helpful to explain this slightly more complicated situation by a simple example.

Example 2 (Decomposing Liveness Properties - Clock Synchronisation) We consider the interaction between two components representing clocks that from time to time send a request signal to receive the actual time from the other clock, say, to refresh their own time. Let us require in the interaction interface description that the time exchange takes place infinitely often. Of course, this can be achieved either by requiring only for one of the components to send infinitely many requests for time signals or by requiring this for both. This shows a case where there are several options to decompose the liveness conditions of the interaction interface into liveness conditions for the components. If this condition is violated we cannot blame only one of the components for this failure.

A liveness condition for both components that is strong enough to guarantee the correctness, is, of course, the requirement for each clock to issue a request signal, provided the other component does not send it up to a certain time point. However, if we know that the other component definitely follows that strategy the one component needs not issue any request signal at all. In other words, in a refinement we either have to work with two specifications of the components that are stronger than needed or we have to work with a design decision introducing an asymmetry between the two components by making only one component responsible for a liveness condition. We can do this by localising the liveness condition for one component when moving from an interaction interface specification to a component interface specification. □

A critical issue in the decomposition of an interaction interface specification into component specifications is, therefore, the decision how to decompose the liveness requirements for the interaction into requirements for the individual components. This actually requires additional design decisions, in general, as illustrated by the example above. To indicate the freedom in the decisions we define the canonical liveness predicate L associated with the interaction interface specification R by the formula

$$L(i,o) \;\equiv\; R(i \oplus o) \vee \neg S(i,o)$$

where S is the safety predicate associated with the interface predicate R defined by the following formula:

$$\begin{aligned} S(i,o) \quad \equiv \quad & \forall t \in \mathbb{N} : \exists i' \in \vec{I}, o' \in \vec{O} : \\ & i{\downarrow}t = i'{\downarrow}t \wedge o{\downarrow}t = o'{\downarrow}t \wedge R(i' \oplus o') \end{aligned}$$

The decomposition of the liveness predicate L into strongest predicates B_Q and B_E for the two components Q and E is quite straightforward. We specify these liveness predicates for Q and E, respectively, by the equations:

$$\begin{aligned} B_Q(i,o) \quad &\equiv \quad (\exists i' \in \vec{I} : L(i',o)) \Rightarrow L(i,o) \\ B_E(i,o) \quad &\equiv \quad (\exists o' \in \vec{O} : L(i',o)) \Rightarrow L(i,o) \end{aligned}$$

However, in general, the predicates B_Q and B_E are unnecessarily strong as a requirement for both components for guaranteeing the liveness condition L. Usually, they can be weakened. In particular, every pair (B'_Q, B'_E) of weaker liveness predicates will do the job provided the following three formulas hold:

$$\begin{gathered} B'_Q(i,o) \wedge B'_E(i,o) \Rightarrow L(i,o) \\ \forall i \in \vec{I} : \exists o \in \vec{O} : B'_Q(i,o) \\ \forall o \in \vec{O} : \exists i \in \vec{I} : B'_E(i,o) \end{gathered}$$

These requirements essentially say that the two liveness conditions B'_Q and B'_E for Q and E guarantee the required global liveness condition included in the interaction interface and for each component at least correct output histories (with respect to liveness) exist.

In general, the choice of the liveness predicates B_Q and B_E obviously leaves freedom to the developer in assigning the responsibility for certain liveness requirements to one of the two components taking part in the interaction. Hence, this step of decomposing liveness requires actually a joint refinement including a design decision, in general, while for the safety properties the responsibilities can be canonically decomposed. When working with independent refinement we have to work with the strongest liveness conditions B_Q and B_E.

6. Refinement of interaction interfaces

For components refinement is simple; it corresponds to the implication relation. We repeat only the most important notions of the refinement relation for components in the following. A component specification F_1 is called a *(property) refinement* of a component specification F_0 if (for all input histories $i \in \vec{I}$ and output histories $o \in \vec{O}$) we have

$$F_1(i).o \Rightarrow F_0(i).o$$

Of course, from a methodological point of view such a refinement only makes sense if it refines a consistent specification into a consistent specification.

6.1. Consistency of Specifications

For a component it is simple to specify consistency. A component specification is called *consistent* if it is realisable. A useful refinement always leads from a consistent specification to a consistent specification. For an interaction interface specification notions of consistency and refinement are not so simple[8]. For components a meaningful refinement of a consistent component specification is required to lead to a consistent component specification that implies the original one. It does not make sense, however, to carry over the definition of consistency for components given above to interaction interface descriptions. Interaction interfaces do only provide scenarios of interaction for those input patterns that we expect to occur in the interaction. If a certain input history does not occur in the interaction, nothing is fixed about the behaviour in this case for a component. Therefore, we derive from an interface specification

$$R : \vec{C} \to \mathbb{B}$$

a component specification

$$F : \vec{I} \to (\vec{O} \to \mathbb{B})$$

as described in the previous section. In general, we do not assume that R is total or time guarded. A component specification has to be total and time guarded, however. We derive a canonical specification for the component F as follows. We require (for all $i \in \vec{I}$):

$$\begin{array}{c} \forall t \in \mathbb{N} : \forall i' \in \vec{I} : i{\downarrow}t = i'{\downarrow}t \Rightarrow \\ (\exists o' \in \vec{O} : R(i' \oplus o') \\ \vee \\ (\forall o \in \vec{O} : F(i).o \Rightarrow \exists o' \in \vec{O} : o'{\downarrow}t{+}1 = o{\downarrow}t{+}1 \\ \wedge\ R(i', o')) \end{array}$$

By this specification we require that for every input i and every time point t for all output histories o we have: for all inputs i' which coincide with i until time t, either i' cannot occur as input ($\neg R(i' \oplus o')$ for all o') or there exists an output o' with $R(i' \oplus o')$ that coincides with o until time $t{+}1$.

6.2. Refinement Notions for Interface Specifications

A simple and obvious way to handle refinement of interaction interfaces is to base it on component refinement. The idea is that both components involved in the interaction are refined either simultaneously or independently. This may reduce the set of possible interaction histories. However, we distinguish between two ways to carry out the development and the refinement of the components:

[8] Strictly speaking, an interaction interface is consistent, if there is at least one allowed interaction history.

1. joint refinement considers a simultaneous refinement of both components involved and therefore simultaneously a refinement of their interaction interface,
2. separated refinement refines one of the components in isolation without knowledge of refinement steps for the other component.

Given two interaction interface specifications (with sets of channels C, I, O with $C = I \cup O, I \cap O = \emptyset$)

$$R_0, R_1 : \vec{C} \to \mathbb{B}$$

we call the specification R_1 an *I-stable refinement* of the specification R_0, if for all communication histories $x \in \vec{C}$ we have the proposition:

$$R_1(x) \Rightarrow R_0(x)$$

and

$$R_0(x) \Rightarrow \exists x' \in \vec{C} : R_1(x') \wedge x'|_I = x|_I$$

and R_1 is time guarded if R_0 is time guarded. An I-stable refinement R_1 keeps all the communication histories in the set of valuations $\vec{I}$ of the channels in the set I which we can find in the interface specification R_0. Hence it allows for an individual refinement of the component Q, where Q is specified as in the previous chapter.

Refinement is generally based on the idea that the specification of the behaviour of a component contains some underspecification. Underspecification means that the behaviour is not uniquely determined but leaves some freedom of choice for the output for a given input. Operationally speaking the behaviour is nondeterministic. For an interaction specification we may define underspecification as follows: the interface specification R is underspecified with respect to the component Q if for certain input histories $i \in \vec{I}$ there are several distinct output histories $o, o' \in \vec{O}$ such that

$$R(i \oplus o) \wedge R(i \oplus o')$$

In other words, for the input i the component Q is free to react by the output history o or by o'. So it is a reasonable refinement step for Q to go from the specification R to an interface specification R' that holds only for channel valuations $i \oplus o$ but not for the valuation $i \oplus o'$. However, such a step has another consequence. After carrying out this step, the output history o' may no longer occur as a communication history in R'. Therefore, this step makes the specification for the component Q stronger but the requirements for the component E weaker since E has no longer to deal with the input history o'.

Whether this idea of refinement is appropriate depends on the way interaction specifications are used in a systematic, methodologically well-defined way. If an interaction

specification defines the way two components interact such that they can be specified and implemented independently, then the concept of refinement makes only sense if it keeps the whole spectrum of possible behaviours for both components. If the interaction specification is to be refined as such in a joint refinement, we may gain more freedom in the refinement.

The interaction interface specification R_1 is called a *general refinement* of the specification R_0 for a set of channels C decomposed into I and O if R_1 is both I- and O-stable. A general refinement is a special case of an independent refinement of both components.

A way to carry out a refinement of interface specifications is zig-zag refinement. In zig-zag-refinement, we start with an I-stable refinement. Then we do an O-stable refinement. Then we repeat an I-stable refinement and so on, until no further refinement is possible or appropriate. This way the refinement of the two components is mutually dependent. This dependency is not surprising. If we restrict the set of possible output histories of one component the set of input to which the other component is supposed to react in a controlled way is reduced. This implies additional freedom for the second component. Zig-zag-refinement corresponds to independent refinement where the two design teams refining E and Q independently exchange their current specification after a while. This leaves room for further refinements. Zig-zag refinement is a special case of joint refinement. If we perform an O-stable refinement, then for the component with input channels I and output channels O we obtain a more liberal safety condition, in general, since certain input may no longer occur and therefore the assumption is strengthened. This way we may relax the requirements for a component (resulting in an "anti-refinement").

7. Realisability

The conditions of time guardedness guarantee the independent choice of input and output. They, however, are not strong enough to guarantee the realisability of the components. The concept of realisability for interaction interfaces is introduced in this section. To guarantee realisability of the component specifications derived from an interaction specification we have to require, in addition, realisability conditions for the components requiring the validity of the liveness predicates B'_Q and B'_E. A component specification F can only be implemented if it is consistent, namely if for every input history i there exists an output history o such that

$$(F.i).o$$

But in addition to consistency we can implement a component (without additional assumptions about the environment) only if there is a way (we speak of a *strategy*) to fulfil the liveness conditions in the interaction between the component and its environment.

We have introduced the notion of realisability only for component specifications so far. Realisability can be defined and has to be studied also for interaction interfaces. We call an interaction interface $R : \vec{C} \to \mathbb{B}$ *realisable* if there exist time guarded functions

$$f : \vec{I} \to \vec{O}$$

$$g : \vec{O} \to \vec{I}$$

such that for $i \in \vec{I}$, $o \in \vec{O}$:

$$f(i) = o \wedge g(o) = i \Rightarrow R(i \oplus o)$$

R is called *fully realisable* if there exist specifications Q and E such that

$$\begin{array}{ll} & R(i \oplus o) \\ \equiv & [\exists f, g : f \in [\![Q]\!] \wedge f(i) = o \wedge g \in [\![E]\!] \wedge g(o) = i] \end{array}$$

This implies that for the component Q (let the decomposition of the channels C into channels I and O be as before), if there exists a time guarded function

$$f : \vec{I} \to \vec{O}$$

such that $f \in [\![Q]\!]$, we have

$$\forall i \in \vec{I} : (\exists o \in \vec{O} : R(i \oplus o)) \Rightarrow R(i \oplus f(i))$$

Then for every feasible input history i for the component Q the function f yields an output history $f(i)$ that fulfils the requirements that are induced by the interaction interface specification R. Realisability essentially means that there exists a strategy for the component specified by Q (and also for E) to react to input messages step by step by output messages such that the infinite stream generated this way fulfils the liveness requirements. Obviously not every interface specification is realisable. This means that we cannot always find an implementation for a component given an interaction interface specification. A typical example would be the following interface specification

$$R(i \oplus o) = (i \neq o)$$

Both component specifications deduced canonically from this interaction interface are not realisable. The reason is that both components cannot predict in the t-th time interval the output of the partner component while preparing their own output of the t-th time interval. However, in this case, there is trivially a joint refinement of R leading to an interaction interface that is realisable. Therefore, in contrast to component specifications the notion of realisability is not monotonic with respect to interaction specification refinement.

8. Conclusion

In the design of distributed systems it is often more suggestive to describe the interaction between the components at least for representative cases before we describe the components themselves. The reason is that the interaction captures the essential idea of a system decomposition. Interaction interfaces support this method of proceeding. A pragmatic description of interaction interfaces used widely in practice are MSCs (see [9]) that are perhaps more suggestive and easier to comprehend than logical predicates. They are more restricted and less expressive, however, than our interaction predicates, since they provide only sample behaviours and not a complete description of all possible behaviours of the components involved. There are some interesting similarities between the interaction interfaces investigated here, and the protocol and actor classes of ROOM [10]. Protocol classes basically serve the purpose of defining syntactically the set of messages that a pair of components may exchange via two of their connecting ports, while actor classes define the behavior, in particular, the communication sequences, exhibited by actors. [10] also suggests to derive protocol classes and actor behavior - at least partially - from scenarios captured during early design phases. However, because the precise relationship between scenarios and complete actor behavior is left unspecified in ROOM, the former's integration into the design process is rather loose. Interaction interfaces, too, entail the syntactic definition of the messages exchanged by the components under consideration. In addition, as we have shown above, it is - to a large extent - straightforward to integrate state-oriented descriptions of component behavior, and interaction interfaces. This stems, in part, from the fact that the latter are not restricted to representing scenarios as (finite) communication sequences; instead, they allow the designer to describe the complete interaction behavior among a set of components. From this we conclude that in order to integrate graphical description techniques for component interaction, such as MSCs, seamlessly into an overall software development process, their syntax and semantics should be defined such that it allows an easy transition from (incomplete) scenarios to complete interaction descriptions. It is a further interesting question, how, given a finite set of instances of interactions, represented for example in terms of MSCs, we can deduce a logical interaction interface specification. This question needs further investigations and experiments.

Acknowledgments

The authors are grateful to Michael von der Beeck, Jan Philipps, and Bernhard Rumpe for fruitful discussions on this topic. We are especially indebted to Max Breitling, who read a draft version of this text and provided valuable comments. Our work was partially supported by the Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen", by the BMBF-project KorSys, and the industrial research project SysLab sponsored by Siemens Nixdorf and by the DFG under the Leibniz program.

References

[1] G. Berry, G. Gonthier, *The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation*, INRIA, Research Report 842, 1988

[2] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language for Object-Oriented Development. Version 1.0*, 1996

[3] M. Broy, *Some algebraic and functional hocuspocus with ABRACADABRA*, Technische Berichte der Fakultät für Mathematik und Informatik, Universität Passau, 1987, MIP-8717, also in: Information and Software Technology 32, 1990, 686-696

[4] M. Broy, A Functional Rephrasing of the Assumption/Commitment Specification Style, Technische Universität München, Institut für Informatik, TUM-I9417, June 1994, Revised and Extended Version to appear in: Formal Methods in System Design

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern Oriented Software Architecture, A System of Patterns*, Wiley, 1996

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns, Elements of Reusable Object-oriented Software*, Addison-Wesley, 1995

[7] GRAPES-Referenzmanual, *DOMINO, Integrierte Verfahrenstechnik*, Siemens AG, Bereich Daten- und Informationstechnik, 1990

[8] I. Jacobson, *Object-Oriented Software Engineering*, Addison-Wesley, ACM Press, 1992

[9] Message Sequence Charts (MSC), *Recommendation Z.120*, Technical report, ITU-T, 1996

[10] B. Selic, G. Gullekson, P. T. Ward, *Real-Time Object-Oriented Modeling*, Wiley, 1994