T U M

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

A Graphical Description Technique for Communication in Software Architectures

Manfred Broy
Christoph Hofmann
Ingolf Krüger
Monika Schmidt

TUM-I9705
Februar 97

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

A Graphical Description Technique for Communication in Software Architectures*

Manfred Broy, Christoph Hofmann, Ingolf Krüger, Monika Schmidt

Institut für Informatik
Technische Universität München
D-80290 München

Abstract

A crucial aspect of the architecture of a software system is its decomposition into components and the specification of component interactions. In this paper we use an enhanced variant of Extended Event Traces [SHB96] as a graphical technique for the description of such component interactions. It allows us to define interaction patterns that occur frequently within an architecture, in the form of diagrams. The diagrams may be instantiated in various contexts, thus allowing reuse of interaction patterns. We present several examples to show the applicability of our notation. In addition, we provide a formal semantics for our graphical notation, based on sets of traces. Furthermore, we compare our approach to connector specifications in WRIGHT [AG94], another description language for component interaction in software architectures.

* This work was sponsored by the *Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie* (BMBF) under the project ‘ENTSTAND’ and by the *Deutsche Forschungsgemeinschaft* (DFG) under the project ‘Bellevue’.

Contents

1 Introduction

Software architecture is considered as one of the keys to modern software technology. Therefore, in recent years software architecture has attracted a lot of attention in computer science research [HHK+96]. Although no satisfactory formal definition of the term software architecture (or architecture, for short) exists to date, most researchers in the field agree on components and their relationships as architectural constituents.

We can describe various aspects of an architecture by further specifying the kinds of components and relationships we are interested in. Such aspects include, for instance, data models, module structures and component distribution. This allows us to restrict our focus to certain architectural views instead of having to deal with the architecture as a whole. An important architectural view is the logical decomposition of a system into interacting components (consider, for instance, objects in an object-oriented application sending messages between each other). This view allows us to either specify or analyze the overall communication protocol of a possibly complex software system.

In this paper, we introduce a graphical notation for the description of component interaction in software architectures. Our notation is based on Extended Event Traces (EETs, [SHB96]), which are similar to Message Sequence Charts [ITU94]. We enhance this notation by additional operators that allow us to describe interaction architectures succinctly. Furthermore, we present a denotational semantics, which is based on traces of system actions. This semantics allows us to analyze the protocols of component interaction specified graphically and to reason formally about them. This formalization is also the basis for the analysis of the compatibility of components with the interaction protocol, and the verification of protocol properties, such as deadlock freedom. However, we do not elaborate on compatibility and verification issues here, nor do we cover methodological questions (like, for instance, in which contexts EETs are best applied, and which interaction protocol properties should be specified by EETs, and which by other techniques, such as predicate calculus). Here, we focus on the graphical notation, and examples of its application.

The remainder of this paper is structured as follows. In Section 2, we introduce our graphical notation, and provide its semantics informally. Then, in Section 3, we give several examples of EET specifications for software architectures to demonstrate applicability of our notation. Section 4 contains the formal syntax and semantics of EETs, as well as examples that indicate how additional properties can be specified by directly working with the semantics. In Section 5, we compare our approach to the one of Allen and Garlan [AG94] for describing the interaction of components. Finally, Section 6 contains our conclusions and directions for further work.

2 EETs for Component Interaction

In this section, we define the graphical notation that we employ for the description of component interaction in software architecture. In Section 2.1, we briefly discuss our motivation for choosing the Extended Event Traces (EETs) of [SHB96] as the basis for our notation. In Section 2.2, we illustrate the syntax and its semantics by means of examples.

2.1 Extended Event Traces

Event traces emerged in the field of telecommunications (see, for instance, [ITU94]) and are now extensively used in various modeling techniques, such as object-oriented analysis and design methods (cf. [Boo94], [BRJ96]), as well as in architecture descriptions (cf. [GHJ+95], [BMR+96]) where they are used to describe examples of object interactions. Because of their simple graphical syntax and their intuitive concepts they are both understandable and easily applicable without requiring extensive training. Often, however, their semantics is not formally defined, which leads to ambiguities in specifications.

In [SHB96], EETs are introduced as a graphical description technique for component interaction, together with a formal semantics. Their appearance is similar to that of Message Sequence Charts. In contrast to the latter, EETs are based on a smaller set of operators, which simplifies their understandability.

Usually, event traces depict exemplary interaction scenarios for a certain set of components. In the area of software architecture, however, we are interested in the set of *all* interaction sequences that may occur during the lifetime of the participating components. Another important requirement for an interaction description technique is that it offers the ability to compose specifications hierarchically, thus reducing the complexity of interaction structures, and offering the possibility to reuse specifications.

To summarize, the important properties and elements of a description technique for component interaction in software architecture are

- a graphical, intuitive and easy to use syntax
- operators for complete interaction descriptions
- operators supporting structuring and reuse of interaction descriptions
- a formal semantics, based on clear concepts.

To address these issues, we use a slightly modified version of EETs for our purposes. We enhance the structuring mechanism of EETs from [SHB96] by providing an instantiation mechanism for interaction descriptions that allows us to adapt EETs to various contexts. Furthermore, we introduce an interleaving operator that enables us to succinctly express EETs in which the order of some events is of no relevance. In the following, we use the abbreviation EET to refer to our modified graphical notation. References to the original definition in [SHB96] are stated explicitly.

In the remainder of this section we describe our graphical notation and explain its semantics informally. For the presentation of the formal semantics of our notation we refer the reader to Section 4.

2.2 Graphical Notation

Every EET has a unique name and consists of a finite set of interacting components that are identified by their component names. Every component that participates in an EET is depicted by a vertical axis (labelled with the component name) representing the lifetime of that component where time advances from top to bottom. In our approach, the component names are regarded as formal parameters of an EET. Thus, EETs can be combined and may be adapted to a new context by substituting their component names. This allows us to reuse interaction descriptions with modified axes labellings in different contexts. An interaction (or event) is indicated by an arrow that is directed from the initiator of the interaction to the destination component. Arrows may be labelled by an event name together with an optional parameter list in parenthesis. No two events are allowed to occur at the same point in time, i.e. at the same vertical position. We assume message transmission to be instantaneous. Figure 1 shows an EET named "*EET_1*" with three components (named A, B and C) and four events (labelled e, f, g and h).

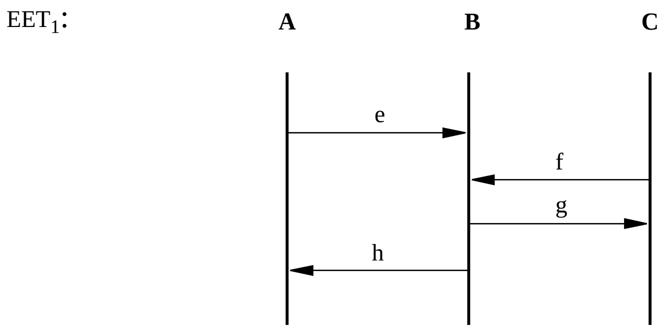


Figure 1 : Simple EET

To reduce the number of EETs needed for an interaction description, option and repetition indicators are provided by the graphical notation. An option indicator (denoted by 0-) allows us to mark parts of an EET as optional, whereas a repetition indicator (-*) denotes parts that may occur repeatedly. Both types of indicators are added to the right of an EET and their vertical expansion designates their scope. Figure 2 depicts an EET where event e is optional, whereas the sequence of messages f followed by g may occur one or more times, before event h occurs.

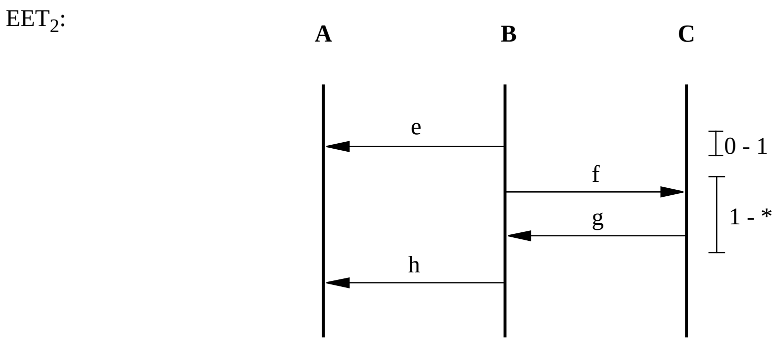


Figure 2 : EET with Optional and Repeated Parts

Furthermore, EETs can be hierarchically structured by a box operator where the box can be instantiated by one of the EETs contained in the set referenced within the box. This set of referenced EETs must be non-empty and finite. The referenced EETs have to be specified elsewhere. This provides a means of maintaining readability in complex EETs.

The meaning of a box referencing a finite set of EETs is a finite choice: when the box is to be unfolded one element from the set has to be chosen. Finite choice allows us to describe alternatives in EETs without having to introduce large numbers of option indicators, while maintaining the intuitive readability of EETs. Note that if finite choice is used in conjunction with a repetition indicator, a different element of the set of possible selections may be chosen in every repetition. EET_3, depicted in Figure 3, denotes the finite set of EETs in which the box named *Choices* is substituted by either EET_1 or EET_2. If the set referenced within a box contains only one element, the latter may be named directly in the box, thus omitting the set completely.

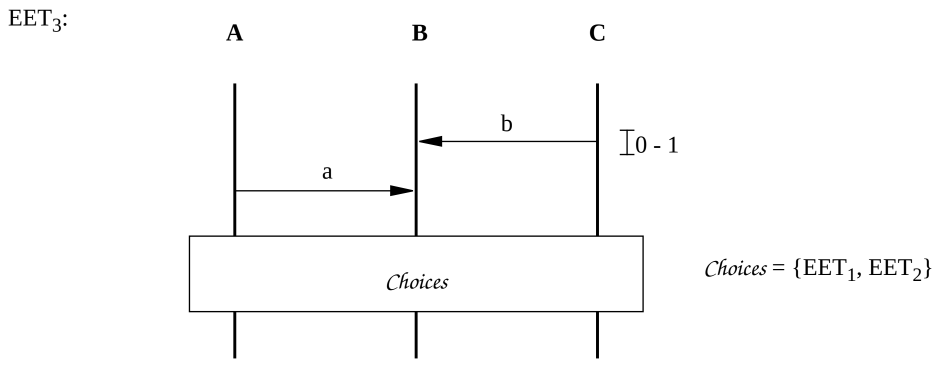


Figure 3 : EET with a Finite Choice Box

We do not allow cyclic or recursive references between EETs; such constructs are used mainly to express repetition, which we handle by the explicit repetition operator. Furthermore, omission of recursive references simplifies understanding of the diagrams, because then, unfolding of the boxes always leads to a finite set of finite diagrams. The original definition in [SHB96] explicitly allows recursive diagrams.

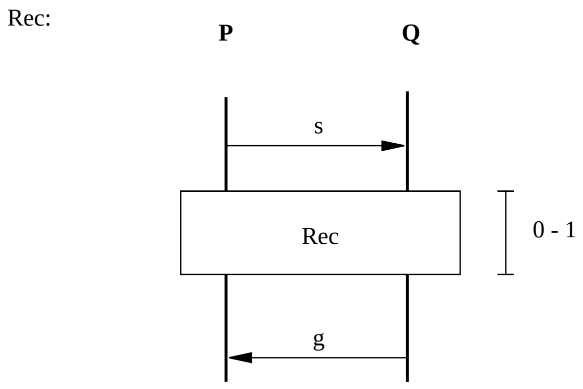


Figure 4 : Recursive EET Definition

If we consider EETs as graphical representations of formal languages the definition given in [SHB96] leads to Chomsky-2-like structures, whereas we restrict ourselves to a Chomsky-3-like language. Consider, for instance, EETs *Rec* and *Iter* of Figure 4 and Figure 5, respectively. Due to its recursive definition *Rec* can be used to specify the following two properties:

1. an equal number of s and g messages have to occur between P and Q, and
2. no g message may precede any s message.

It is not possible to specify both properties graphically in our restricted notation. *Iter*, which complies to our notation, specifies that any positive number of s messages may occur before any positive number of g messages. The equality of the numbers of the respective messages cannot be addressed graphically. However, in Section 4 we show how to overcome this limitation by introducing predicates that specify additional properties of the EET, like, in our example, the equality of the number of occurrences of s and g messages.

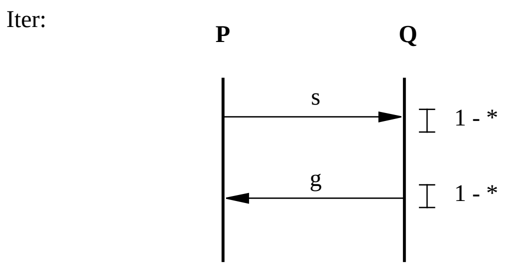


Figure 5 : Nonrecursive EET Definition

As we treat the component names in an EET as formal parameters, EETs can be adapted to a different context, i.e. to different component names. This adaption is expressed by adjoining to the name of the adapted EET the list of new component names so that every (formal) component name of the adapted component (in the labelling order of its axes from left to right) is substituted by the component name of the new context. For instance, *EET_2[D, E, F]* is the same EET as *EET_2* in Figure 2 with the component names A, B and C substituted by the component names D, E and F, respectively. Of course, the compatibility concerning the arity and the naming of the axes of an EET corresponding to a box with its “parent” EET has to be ensured.

Consider, for instance, *EET_4* in Figure 6, where the EETs referenced in set *Choices* are adapted to their new context.

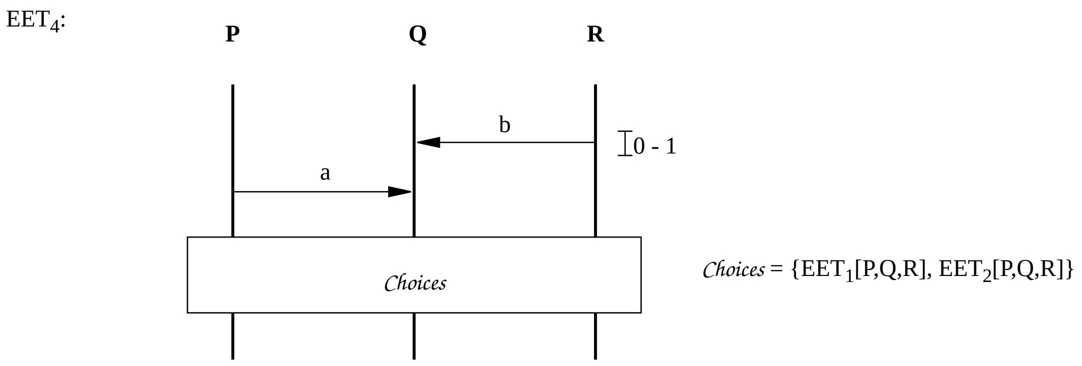


Figure 6 : Adaption of Context

If substitution is applied to a hierarchically structured EET, the application is propagated all the way down the hierarchy. Note that in the original definition in [SHB96] EETs could not be adapted to different contexts explicitly.

Finally, we introduce an operator to denote the interleaving of EETs. Consider, for instance, *EET_5* in Figure 7, which depicts the interleaving of *EET_6* and *EET_7*. Intuitively, this means that events a, b, c and d may occur in any order, provided that b never occurs before a and d never occurs before c. The interleaving operator is an extension of the original EET definition in [SHB96].

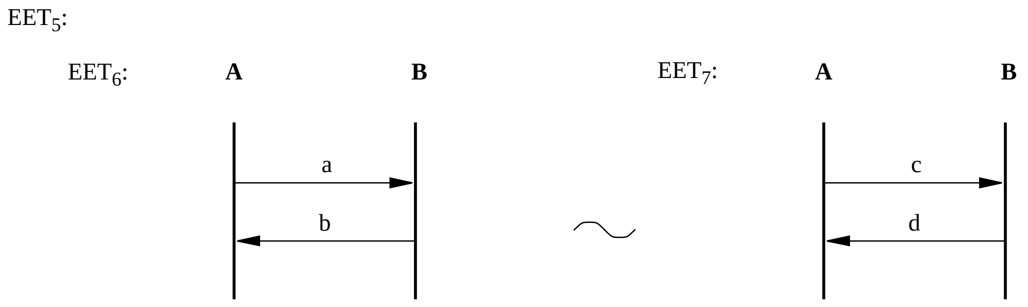


Figure 7 : Interleaving Operator

Intuitively, the semantics of an EET is the set of traces obtained by recording all events while following all possible paths through the graph from top to bottom.

3 EETs for Example Architectures

In this section, we give four examples of interaction architecture specifications using EETs. These examples demonstrate the practical use of all operators introduced in Section 2.2. Some examples are inspired by [AG94] such that it is easier to compare the two description techniques later on (see Section 5).

3.1 Client/Server

The first example shows how the interaction of components in a simple Client/Server system that consists of exactly one client and one server can be modelled. The system is then extended by additional clients communicating with the same server. The interaction of a single client and a server component can be described easily by the EET in Figure 8.

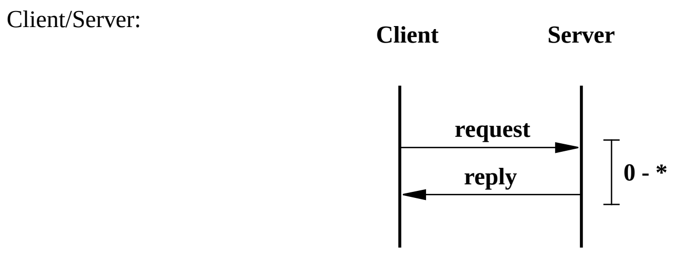


Figure 8 : EET for the Client/Server Interaction Architecture

The client sends a request message to the server. This message is followed by a reply message in the opposite direction. This message sequence may occur never, or may be repeated finitely often.

Now, we want to extend this example by an additional client that communicates with the same server. The communication behavior of each client is as described in the EET above. We assume that the server is able to process the requests in parallel. After completely processing one client's request, the server sends back a reply message to that client. The EET describing the explained interaction architecture is depicted in Figure 9.

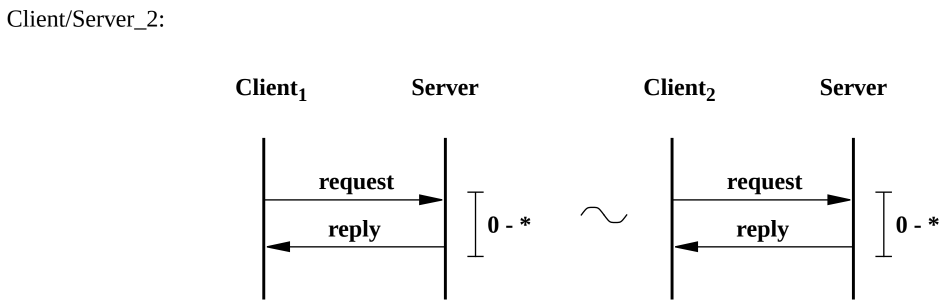


Figure 9 : EET for the Interaction of two Clients with a Single Server

The EET in Figure 9 describes all system traces where request and the corresponding reply messages may be interleaved with two restrictions resulting from each operand EET of the interleaving operator. These restrictions are:

- Between any two request messages sent by a specific client, that client has to receive a reply message from the server.
- The server sends a reply message to a client only after having received a request message from that client.

If a system with one server and n ($n \in \mathbb{N}$) clients is considered (where the interaction of each client with the server is described as in Figure 8), the interaction architecture would be described by the interleaving of n EETs.

This example shows how EETs may be used to specify properties of a communication architecture elegantly. This is in contrast to other formal specification techniques, such as predicate calculus, where formulation of such properties usually requires a substantial amount of work.

3.2 Shared Variable Access

This example allows us to show the use of the finite choice operator in combination with axes renaming. The mentioned system consists of two users sharing a common variable. First of all, one of the users has to perform the initialization. Afterwards both of them can read or change the value of the variable without any restrictions. These two scenarios are separately described by the two EETs depicted in Figure 10. EET *Set* represents the event that occurs when the shared variable is set to some value; EET *Get* describes the message exchange when reading the shared variable's value.

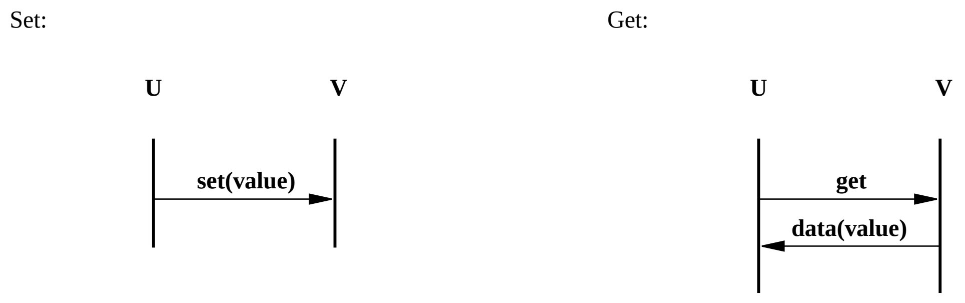


Figure 10 : Variable Access

To describe the behavior of two users accessing the shared variable, we use EET block specifications here, and instantiate them to the special context of the example by renaming their axes as shown in Figure 11.

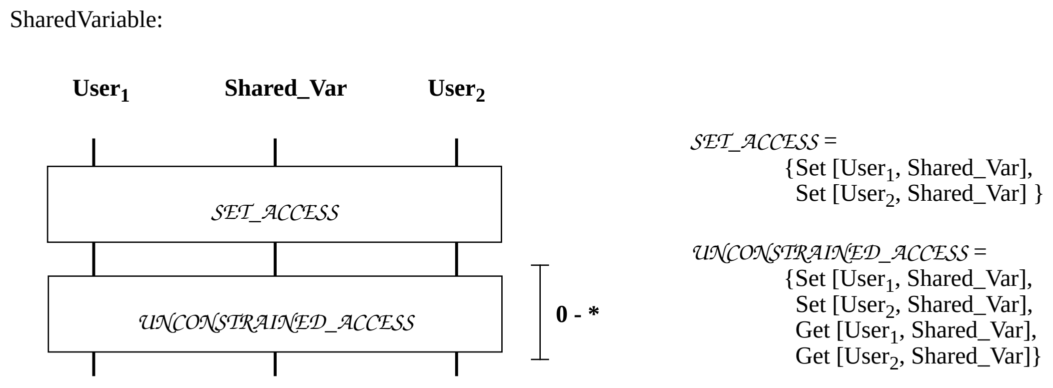

SharedVariable:
User1
Shared_Var
User2
SET_ACCESS
UNCONSTRAINED_ACCESS
0 - *
SET_ACCESS = {Set [User1, Shared_Var], Set [User2, Shared_Var] }
UNCONSTRAINED_ACCESS = {Set [User1, Shared_Var], Set [User2, Shared_Var], Get [User1, Shared_Var], Get [User2, Shared_Var]}

Figure 11 : Shared Variable Access

Figure 11 describes all EETs resulting from the concatenation of EETs starting with one element of the set *SET_ACCESS* followed by an optional and finite sequence of elements of *UNCONSTRAINED_ACCESS*.

This example shows the advantage of using the finite choice operator. Without it we would have to draw an EET for all combinations of alternatives specified in the two EET sets *SET_ACCESS* and *UNCONSTRAINED_ACCESS* of Figure 11. This would result in a large number of EETs. Therefore, boxes and the finite choice operator are an elegant notation allowing us to represent the complete interaction architecture of a system with exactly one EET.

In most cases, boxes represent interaction patterns that are typical for the system under consideration. Therefore, by use of boxes EETs become more structured and are easier to read and understand.

3.3 Observer Pattern

Patterns [GHJ+95] [BMR+96] provide an intuitive, albeit informal, presentation technique for (parts of) software architectures. Pattern descriptions consist of a problem statement and the solution to the problem in a certain context. Usually, interaction scenarios of the components participating in the solution are graphically described by event traces. Here, we will demonstrate applicability of our notation for that purpose.

As an example pattern, we consider Observer [GHJ+95]. Observer presents a solution to the following problem: Given a component whose state changes frequently, and other components that are dependent on this state. How can the latter's states be kept consistent with the former's? An application of the Observer pattern can be found in [BMR+96], where it forms the basis for the well-known Model-View-Controller architecture.

The structure of the solution, depicted as a class diagram using an OMT-like notation [RBP+91], is given in Figure 12.

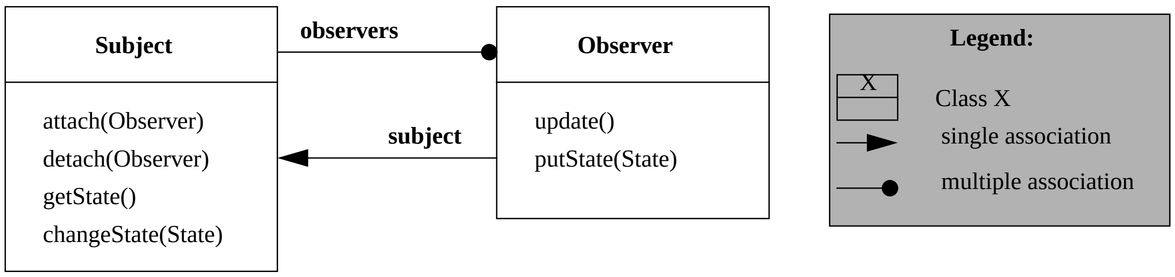


Figure 12 : Simplified Structure of the Observer Pattern

In [GHJ+95], the authors present a more general solution using inheritance. We focus on the interaction scenario, hence we deal with this simpler version.

The participants in this architecture are Subject and a number of Observers. Every Observer that wants to receive notification of state changes in the Subject registers with the latter by sending message attach. To decouple from its Subject an Observer sends message detach. Whenever the Subject's state is changed via changeState, all registered Observers are notified by an update event. They may then request to receive the updated state by sending message getState, which causes Subject to return a putState event.

We now specify this interaction architecture graphically. We focus on the update mechanism and omit the registration and decoupling operations. For this example, we assume that the number of Observers that have attached to a single Subject is n, $n \in \mathbb{N}$. Two basic interaction scenarios are depicted by EETs in Figure 13. The first one (*StateChange*) describes the sending of message changeState from an Observer to the Subject, the second one (*SingleUpdate*) expresses the notification of an Observer and the subsequent request for and transmission of the changed state between Observer and Subject.

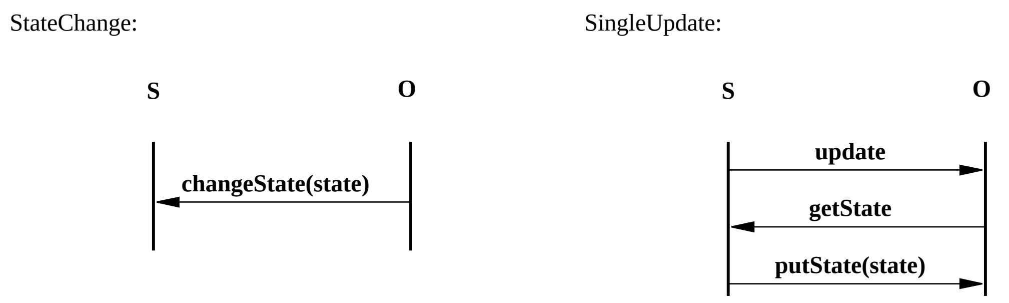


Figure 13 : State Change of the Subject and State Transmission to a Single Observer

Note that, as described by *SingleUpdate*, the Observer has to request and receive the updated state. By means of an option indicator we could easily specify the state request and reply mechanism as optional for the Observer, thus yielding a variant of the pattern. Next, we describe the notification of all Observers by a single Subject. We use the interleaving operator to denote that the order in which the Observers are notified does not matter (cf. Figure 14).

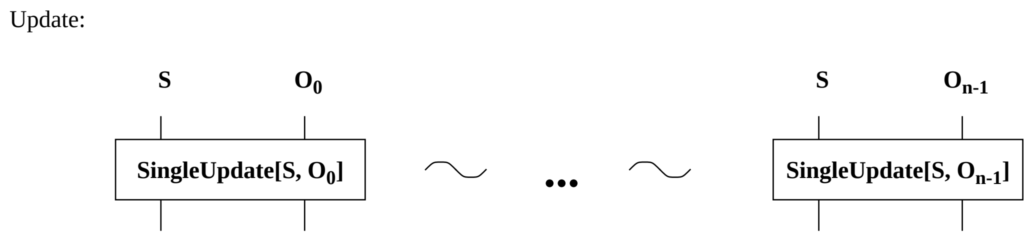


Figure 14 : Updating the State Information of all Registered Observers O_0, ..., O_{n-1} with Subject S

Now, we combine EETs *StateChange* and *Update* to yield a communication architecture for the Observer pattern, where state changes are requested by one Observer at a time, followed by an update broadcast to all registered Observers (cf. Figure 15).

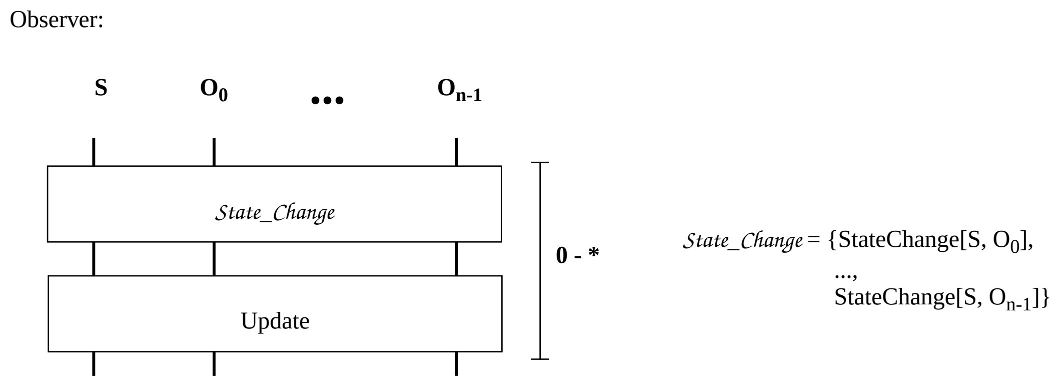


Figure 15 : Observer Interaction Architecture after Attachment of Observers O_0, ..., O_{n-1} with Subject S

Note that the interaction architecture, as depicted in Figure 15, forbids interleaved sending of changeState requests by different Observers. Instead, it specifies that first the complete update cycle is processed, before another changeState request is handled by the Subject. Of course, other request handling strategies could be specified as well.

This example, again, demonstrates that our graphical notation allows us to specify interaction architectures including all participating components during the lifetime of the system, which is an interesting extension of the example scenarios presented in [GHJ+95] and [BMR+96]. We will return to this example in Section 4.4.1.

3.4 Pipe

The last example we study is a more complex one specifying a pipe architecture with a writer that writes messages to the pipe and a reader that reads messages from the pipe. The interaction with the pipe can be closed independently by the writer or the reader. If the reader gets an eof_msg message, which appears when the writer has closed its connection, then the reader has to close its connection eventually, as well. If, on the other hand, the reader decides to close the pipe the writer can continue writing to the pipe.

Again, we specify the interaction architecture by typical interaction patterns. Figures 16 to 18 depict four EETs, each representing such a pattern.

EET *Write* (see Figure 16) shows the message the writer sends to the pipe in order to write a value to the pipe. *Read* describes the interactions taking place when the reader reads a value from the pipe. First the reader has to send a read message and then receives a message from the pipe that contains the value as a parameter.

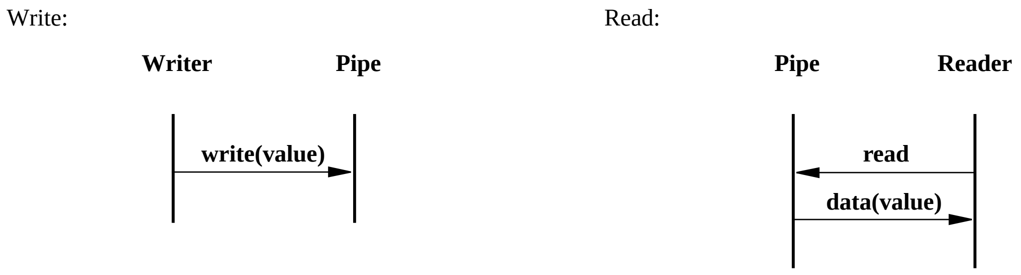


Figure 16 : Read and Write Scenario

Figure 17 describes the situation when the writer closes its connection to the pipe without the reader having closed the connection until now. First the writer sends a close message to the pipe, then the reader can read from the pipe until the latter sends an eof_msg. Afterwards the reader has to close the connection as well.

WriterCloses:

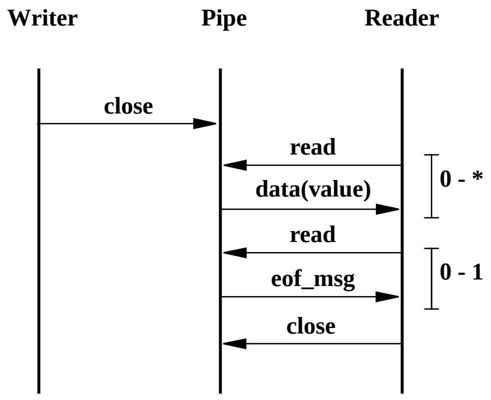


Figure 17 : “Writer Closes” Scenario

If the reader component is the first one that closes the connection, the writer may send write(value) messages to the pipe until it closes the connection as well (see Figure 18).

ReaderCloses:

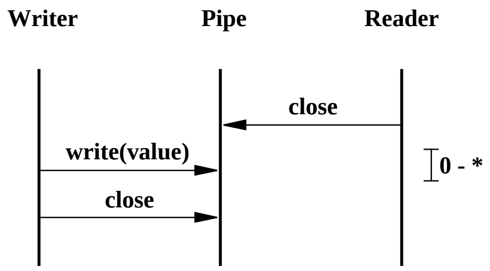
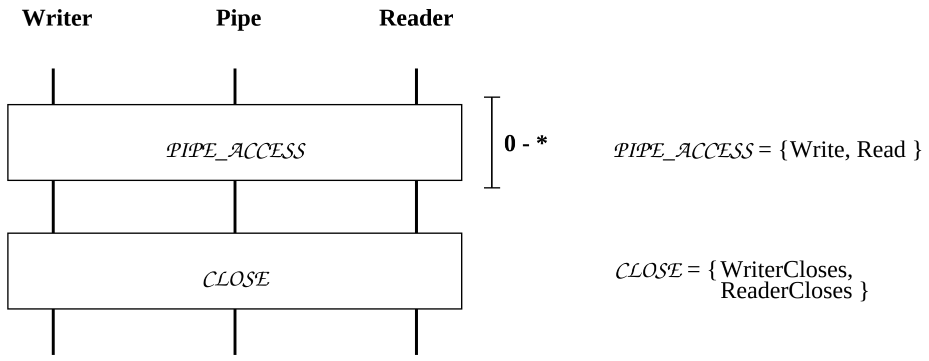


Figure 18 : “Reader Closes” Scenario

To describe the behavior of the pipe interaction architecture, these EETs are composed to form the EET of Figure 19. Note that we did not have to perform a context adaption in *Pipe* because the referenced EETs already have the right context.

Pipe:



Writer
Pipe
Reader
PIPE_ACCESS
0 - *
PIPE_ACCESS = {Write, Read }
CLOSE
CLOSE = { WriterCloses, ReaderCloses }

Figure 19 : Pipe Interaction Architecture

Looking at the EET depicted in Figure 19, the structure of the system behavior is obvious: after an initial phase of writing to and reading from the pipe repeatedly, either the writer or the reader closes the pipe (one element of $\mathcal{CLOSE}$).

Note that we cannot specify the FIFO property, which a pipe typically has, in our graphical notation. Our trace semantics, which we introduce in Section 4, allows us to formulate predicates over traces. Such predicates may be used to restrict the set of traces described by an EET, thus yielding only the traces with the desired properties (such as FIFO).

4 Formal Syntax and Semantics

In the preceding sections we have used our variant of EETs based on the informal semantics given in Section 2.2. However, the graphical notation has limitations with respect to its expressive power. Consider, for instance, the FIFO property of the pipe architecture described above. This property cannot be specified graphically in our notation. To remedy this situation we define a formal semantics for our variant of EETs that is based on sets of streams. To specify additional properties of systems described by EETs we use predicates over these streams. This allows us to formulate, for instance, the FIFO property in an elegant way.

In Section 4.1, we present a formal syntax for a simple language that can be used as a textual representation for EETs. We also define a denotational semantics for that language in Section 4.2, based on predicates over streams. Then, in Section 4.3, we establish the relationship between the graphical and the textual representation. To illustrate the applicability of our semantics we present two examples in Section 4.4.

The presentation in this section has been strongly influenced by [Fac95], where a similar approach is pursued for Time Sequence Diagrams [ISO94], which are used for the specification of OSI services.

4.1 Textual Representation

The first step towards the definition of a formal semantics for EETs is to define a textual representation. It is given by the following BNF grammar:

```
<EETL>  ::=   empty
         |    <Msg>
         |    <EETL> ; <EETL>
         |    <EETL> | <EETL>
         |    <EETL> ~ <EETL>
         |    <EETL>*
```

The syntactic category <Msg> is used to represent the messages that constitute the interactions between components.

```
<Msg>   ::=   ( <ComponentName> , <ComponentName> , <MsgH> )
<MsgH>  ::=   <MessageName>
         |    <MessageName> , ( <FPars> )
```

<FPars>	::=	<FParName>
	|	<FParName> **,** <FPars>

The productions for <ComponentName>, <MessageName> and <FParName> are omitted here. They can be represented by strings denoting component, message and formal message parameter names, respectively. The component names in a message reference the sender and the receiver of the message in that order. Thus, a message determines its sender and receiver, the message name and an optional parameter tuple.

For reasons of brevity, in the sequel we identify nonterminals and the languages they generate.

Parenthesis may be used to group <EETL> expressions. To increase readability we define the following precedence rule: All operators associate to the left. The operators **;**, **|** and **~** bind equally strong. ***** has highest precedence, i.e. all other operators have lower precedence than *****.

The intuitive interpretation for the operators defined in the productions of <EETL> will be sequential composition, alternative, interleaving and repetition, respectively. This set of operators has been chosen to allow us a smooth transition from the graphical to the textual representation of EETs.

Next, we define a formal semantics for the language introduced in this section.

4.2 Denotational Semantics for <EETL>

The denotational semantics of <EETL> is defined by predicates over streams. A stream over a set M is a finite or infinite sequence of elements from M. By M^* and M^∞ we denote the set of finite and infinite sequences of elements from M, respectively. Then $M^\omega = M^* \cup M^\infty$ is the set of streams over M. The empty stream is denoted by ε and the powerset of M is denoted by $\wp(M)$.

Without formal definition, we use the following operators on streams over set M:

&	$: M \times M^\omega \to M^\omega$
#	$: M^\omega \to \mathbb{N} \cup \{\infty\}$
<.>	$: M \to M^\omega$
$\circ$	$: M^\omega \times M^\omega \to M^\omega$
©	$: \wp(M) \times M^\omega \to M^\omega$
ft	$: M^\omega \to M \cup \{\perp\}$
rt	$: M^\omega \to M^\omega$
$\sqsubseteq$	$: M^\omega \times M^\omega \to \mathbb{B}$

By $\mathbb{B}$ = {true, false} and $\mathbb{N}$ we denote the set of Boolean truth values and the set of natural numbers, respectively. For $m \in M$, $s, t \in M^\omega$ and $N \subseteq M$ the purpose of these operators can be described as follows: ft.s yields the first element in s, or $\perp$ if $s = \varepsilon$. Stream s without its first element is denoted by rt.s where $rt.\varepsilon = \varepsilon$. m&s yields the stream whose first element is m and then continues as s. #s denotes the length of (i.e. the number of elements in) s. If $s \in M^\infty$ then

$\#s = \infty$. The term $<m>$ denotes the stream consisting of only the message m. $s \circ t$ yields the stream obtained by prepending stream s to stream t. If $s \in M^\infty$ then $s \circ t = s$. $N©s$ yields the stream obtained from s by removing all elements not in N. $\sqsubseteq$ denotes the prefix ordering on streams. We write $s \sqsubseteq t$ if s is a prefix of t. For a formal definition of these operators see, for instance, [Fac95].

Furthermore, we define a canonical extension of functions mapping a message set M to another message set N: For each function

$$f \;: M \rightarrow N$$

we define its corresponding extension to streams by

$$f^\omega : M^\omega \rightarrow N^\omega$$

$$f^\omega.\varepsilon = \varepsilon$$
$$f^\omega.(m\&t) = f.m \;\&\; f^\omega.t.$$

For better readability we use f.x instead of f(x) to denote application of function f to argument x.

The basic idea of our denotational semantics is that every <EETL> expression defines a predicate that describes the sequence of interactions between the participating components. Once we have such a predicate we can define the set of streams that describes all possible interaction sequences as given by the <EETL> expression.

As a representation for the syntactic category <Msg>, we define the following set:

$$Msg = CN \times CN \times MN \times (\mathbb{N} \rightarrow PN)$$

By CN, MN and PN we denote the sets of component, message and message parameter names, respectively. A message consists of the name of its source and the destination component, a message name and a parameter tuple. Note that we impose the following restriction on the parameter tuple: its domain has to be an interval of the form [0 .. i], for $i \in \mathbb{N}$. Thus, we consider only partial mappings from $\mathbb{N}$ to PN, whose domain is an interval of the given form, to be valid parameter tuples. The parameter tuple may be omitted if it is empty.

We use function

$$M[\![.]\!] : <Msg> \rightarrow Msg$$

to denote the mapping between the syntactic category <Msg> and its denotation. Its obvious definition is skipped here.

When defining the semantics of <EETL> expressions we have to take into account that <EETL> only deals with messages with formal parameter names. An actual interaction pattern consists of messages where these formal parameters are substituted by some actual values. Therefore, we define the set AMsg denoting the set of actual messages as follows:

$$AMsg = CN \times CN \times MN \times (\mathbb{N} \rightarrow VAL)$$

The set VAL denotes the set of all values that can be substituted for formal parameters in messages. Again, we impose the restriction on the parameter tuple that it be a partial mapping from $\mathbb{N}$ to VAL, whose domain is an integer interval starting at 0. For convenient access to the constituents of an actual message, we define the following functions:

sender	$: AMsg \rightarrow CN$
receiver	$: AMsg \rightarrow CN$
msgName	$: AMsg \rightarrow MN$
paramValues	$: AMsg \rightarrow (\mathbb{N} \rightarrow VAL)$

by

$$m \in AMsg \wedge m = (s, r, mn, pv) \Rightarrow \quad sender.m = s \wedge receiver.m = r \wedge msgName.m = mn \wedge paramValues.m = pv$$

Furthermore, for every formal message m’ ∈ Msg we assume a mapping

$$dom: Msg \rightarrow \wp(AMsg)$$

such that dom.m’ denotes the set of all possible messages where the formal parameters are replaced by concrete values.

Now, we apply the operators introduced above for the formal definition of a semantics for <EETL>. We use induction over the structure of the grammar. To that end, we define the following function:

$$P[\![.]\!] : \langle EETL \rangle \rightarrow (AMsg^* \rightarrow \mathbb{B})$$

Intuitively, for any $\alpha \in$ <EETL>, $P[\![\alpha]\!]$ yields a predicate that describes the set of streams generated by α. Let $\alpha, \beta \in$ <EETL>, $m \in$ <Msg> and $t \in AMsg^*$. We define $P[\![.]\!]$ as follows:

$$P[\![\textbf{empty}]\!].t \equiv (t = \varepsilon)$$

Thus, **empty** generates the empty stream.

$$P[\![m]\!].t \equiv (\exists m' : m' \in dom.M[\![m]\!]: t = \langle m' \rangle)$$

A single message generates the set of streams consisting of exactly one message where all the formal parameters have been substituted by actual values.

$$P[\![\alpha \,\mathbf{;}\, \beta]\!].t \equiv \quad (\exists s_0, s_1 : \quad s_0 \in AMsg^* \wedge s_1 \in AMsg^* : t = s_0 \circ s_1 \wedge P[\![\alpha]\!].s_0 \wedge P[\![\beta]\!].s_1)$$

This defines **;** to denote sequential composition of <EETL> expressions.

$$P[\![\alpha \mid \beta]\!].t \equiv (P[\![\alpha]\!].t \vee P[\![\beta]\!].t)$$

Due to this definition **|** denotes the choice operator for <EETL> expressions.

$$
\begin{aligned}
P[\![\alpha \sim \beta]\!].t \equiv\; & (\exists bs: bs \in \mathbb{B}^{\infty} : P[\![\alpha]\!].(f.true.bs.t) \wedge P[\![\beta]\!].(f.false.bs.t)) \\
& \text{where} \quad f : \mathbb{B} \times \mathbb{B}^{\infty} \times AMsg^{*} \to AMsg^{*} \text{ is defined by} \\
& \quad f.x.y.\varepsilon = \varepsilon \\
& \quad f.x.(y_0\&y).(s_0\&s) = \textbf{if } (x = y_0) \\
& \qquad\qquad \textbf{then } s_0\&f.x.y.s \\
& \qquad\qquad \textbf{else } f.x.y.s
\end{aligned}
$$

Thus, $\alpha \sim \beta$ denotes the (not necessarily fair) interleaving of the streams generated by α and β, respectively.

$$
\begin{aligned}
P[\![\alpha^{*}]\!].t \equiv\; & (\mu Z.(\tau.P[\![\alpha]\!].Z)).t \\
& \text{where} \quad \tau : (AMsg^{*} \to \mathbb{B}) \to (AMsg^{*} \to \mathbb{B}) \to (AMsg^{*} \to \mathbb{B}) \\
& \text{is defined by} \\
& \tau.E.A.s \equiv (\quad s = \varepsilon \\
& \qquad \vee\ (\exists s_0, s_1 : s_0 \in AMsg^{*} \wedge s_1 \in AMsg^{*} : \\
& \qquad\qquad s_0 \neq \varepsilon \wedge s = s_0 \circ s_1 \wedge E.s_0 \wedge A.s_1))
\end{aligned}
$$

Here, μZ denotes the least fixpoint operator. τ is the functional that generates predicates describing all finite repetitions of the stream generated by α. Existence of the least fixpoint is guaranteed by the monotonicity of τ in its second argument according to the theorem of Knaster-Tarski.

The definition of $P[\![.]\!]$ as given above allows us to assign a semantics to all <EETL> expressions. It is the set of streams generated by the expression under $P[\![.]\!]$.

$$[\![\ .\ ]\!] : \langle EETL \rangle \to \wp(AMsg^{*})$$

$$[\![\alpha]\!] = \{t \in AMsg^{*} : P[\![\alpha]\!].t\}$$

As we might constrain the set of possible system traces by some further predicate, we define the following semantics $[\![.\ ,\ .]\!]$ for an <EETL> expression and a constraining predicate by:

$$[\![.\ ,\ .]\!] : \langle EETL \rangle \times (AMsg^{*} \to \mathbb{B}) \to \wp(AMsg^{*})$$

$$[\![\alpha, Q]\!] = \{t \in [\![\alpha]\!]: Q.t\}$$

Now that we have defined a formal semantics for <EETL>, we relate EETs and <EETL> expressions. Thereby, we obtain a formal semantics for EETs.

4.3 Relating EETs and <EETL> Expressions

Due to the set of operators and the semantics we have defined for <EETL> expressions it is straightforward to obtain an <EETL> expression from any given EET. The sets CN, MN and PN can be derived from the names of the components, messages and message parameters, respectively, as referenced in the EET. We assume that all referenced EETs are properly adapted to the general EET context. In the following, we provide a function $\mathcal{T}$ from EET diagrams to <EETL> expressions by describing its transformation rules that we can apply to the different structures of EET diagrams:

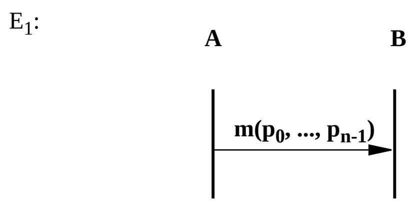


Figure 20 : Single Message

The sending of a message between components A and B, as depicted in Figure 20, is transformed into the following textual representation:

$$\mathcal{T}(E_1) = (\mathrm{A}, \mathrm{B}, \mathrm{m}, (p_0, ..., p_{n-1}))$$

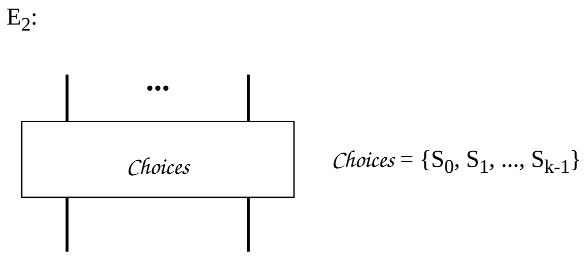


Figure 21 : Choice Operator

The concept of finite choice in EETs can be modelled by application of the alternative operator of <EETL>. Hence, the textual representation of the graph depicted in Figure 21 is defined by

$$\mathcal{T}(E_2) = \mathcal{T}(S_0) \mid \mathcal{T}(S_1) \mid ... \mid \mathcal{T}(S_{k-1})$$

We demand that all names of the axes of all EETs S_i for $0 \leq i \leq k\text{-}1$ are also names of axes in EET E_2.

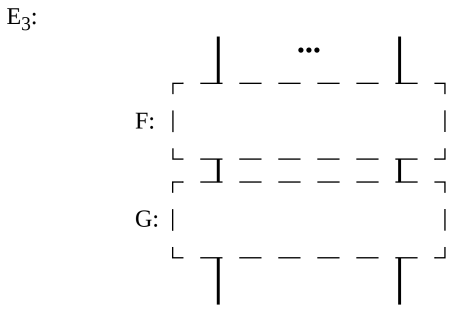


Figure 22 : Sequence of EETs

Putting one EET after another is transformed into the sequential composition of their textual representations. Thus, the graph depicted in Figure 22 is translated into:

$$\mathcal{T}(E_3) = \mathcal{T}(F) \; \mathbf{;} \; \mathcal{T}(G)$$

Each dotted box in this figure denotes any EET with compatible axes names.

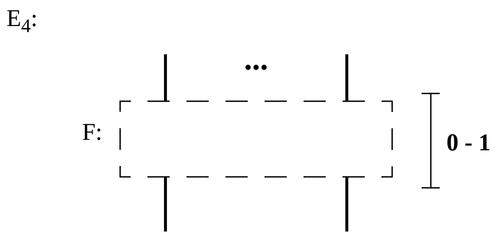


Figure 23 : Indicator 0 - 1

The indicator 0-1 is transformed by application of the choice operator for <EETL>. One of the operands is the <EETL> expression **empty**. The graph of Figure 23 thus yields the following textual representation:

$$\mathcal{T}(E_4) = \textbf{empty} \mid \mathcal{T}(F)$$

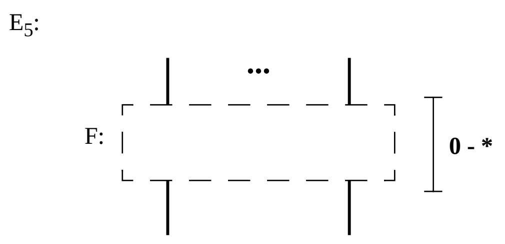


Figure 24 : Indicator 0 - *

Indicator 0-***** is directly translated into application of operator *****. The EETL expression of the EET in Figure 24 is given by:

$$\mathcal{T}(E_5) = \mathcal{T}(F)^{*}$$

Other indicators, like 1-*, can also be translated easily.

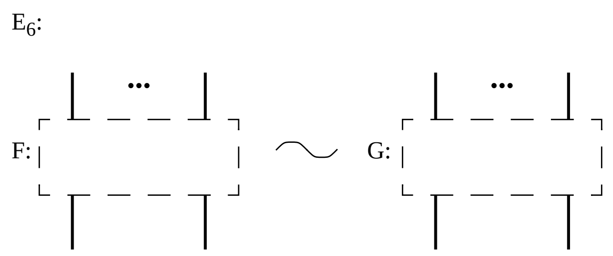


Figure 25 : Interleaving Operator

Two EETs, used to describe independent interaction patterns between components, can be translated by application of the interleaving operator of <EETL>. The EET depicted in Figure 25 may be transformed according to:

$$\mathcal{T}(E_6) = \mathcal{T}(F) \sim \mathcal{T}(G)$$

This translation may be formalized further in a straightforward manner. For reasons of brevity we do not go into the details here.

4.4 Examples

In the following we provide two examples where we employ the formal semantics to add properties to interaction architectures specified in our graphical notation. In Section 4.4.1 we continue our treatment of the Observer pattern from Section 3.3 by adding the property that within an update cycle every Observer receives the same state. Section 4.4.2 contains a formal specification of the FIFO property for the Pipe architecture of Section 3.4.

4.4.1 Observer Pattern

In our presentation of the interaction architecture of the Observer pattern in Section 3.3 we could not state formally that within one update cycle the states transmitted from the Subject to the Observers are equal. In presentations of the Observer pattern, such as [GHJ+95], this “kind behavior” of the Subject is usually taken for granted. Here, however, we formalize this property.

In order to be able to formulate properties for EET *Observer* (cf. Figure 15) that describes the interaction architecture of the Observer pattern, we first have to translate the graphical representation of the EET into a corresponding <EETL> expression. According to the transformation rules given in Section 4.3, we obtain the following textual representations for the EETs depicted in Figure 13 to Figure 15:

$$\begin{array}{lcl}
\mathcal{T}(\mathit{StateChange}) & = & (\text{O, S, changeState, (state)}) \\[1ex]
\mathcal{T}(\mathit{SingleUpdate}) & = & (\text{S, O, update}); (\text{O, S, getState}); (\text{S, O, putState, (state)}) \\[1ex]
\mathcal{T}(\mathit{Update}) & = & \quad ((\text{S}, \text{O}_0, \text{update}); (\text{O}_0, \text{S, getState}); (\text{S}, \text{O}_0, \text{putState, (state)})) \\
 & & \sim\ ((\text{S}, \text{O}_1, \text{update}); (\text{O}_1, \text{S, getState}); (\text{S}, \text{O}_1, \text{putState, (state)})) \\
 & & \sim\ \ldots \\
 & & \sim\ ((\text{S}, \text{O}_{n-1}, \text{update}); (\text{O}_{n-1}, \text{S, getState}); (\text{S}, \text{O}_{n-1}, \text{putState, (state)})) \\[1ex]
\mathcal{T}(\mathit{Observer}) & = & (\ (\quad\ \ (\text{O}_0, \text{S, changeState, (state)}) \\
 & & \qquad\ \mid\ (\text{O}_1, \text{S, changeState, (state)}) \\
 & & \qquad\ \mid\ \ldots \\
 & & \qquad\ \mid\ (\text{O}_{n-1}, \text{S, changeState, (state)})); \\
 & & \quad \mathcal{T}(\mathit{Update}))^*
\end{array}$$

To obtain $\mathcal{T}(\mathit{Update})$ and $\mathcal{T}(\mathit{Observer})$ we have performed the necessary substitutions in the textual representations of *SingleUpdate* and *StateChange*, respectively. Furthermore, for reasons of readability, we have used $\mathcal{T}(\mathit{Update})$ in the definition of $\mathcal{T}(\mathit{Observer})$.

The semantics of $\mathcal{T}(\mathit{Observer})$, i.e. $[\![\mathcal{T}(\mathit{Observer})]\!]$, contains all message streams where change requests by some O_i, $0 \leq i \leq n$, are followed by sequences of update messages (containing arbitrary state values) between S and all O_j, $0 \leq j \leq n$. Now, we will restrict this trace set further to ensure that within one update cycle all Observers receive the same state value. To that end, we define the following predicate for all $t \in AMsg^*$:

$$\begin{array}{lll}
P_{\text{StateConsistent}}.t \equiv & t = \varepsilon & \\
 & \vee\ (\forall\, t_0, t_1: & t_0 \in AMsg^* \wedge t_1 \in AMsg^* \wedge t_1 \neq \varepsilon : \\
 & & (\quad (t_0 \circ t_1) \sqsubseteq t \\
 & & \quad \wedge\ \text{msgName}.(\text{ft}.t_1) = \text{changeState} \\
 & & \quad \wedge\ \#(M_{\text{changeState}} \copyright\ \text{rt}.t_1) = 0) \\
 & & \Rightarrow \text{equalValues}.(\text{paramValues}^{\omega}.(M_{\text{putState}} \copyright\ \text{rt}.t_1)))
\end{array}$$

In the definition of this predicate we have used the following abbreviations:

$$M_{\text{changeState}} = \{m \in AMsg : \text{msgName}.m = \text{changeState}\},$$

$$M_{\text{putState}} = \{m \in AMsg : \text{msgName}.m = \text{putState}\}$$

Predicate

$$\text{equalValues} : VAL^* \rightarrow \mathbb{B}$$

yields true iff all elements of the given stream of parameter values are equal. Its obvious definition is omitted for reasons of brevity.

Intuitively, $P_{\text{StateConsistent}}.t$ states that in any subsequence of t that starts with a changeState message, all parameter values of the subsequent putState messages are equal until the next changeState message occurs.

Predicate $P_{StateConsistent}$ allows us to define the semantics of the Observer interaction architecture we considered in Section 3.3 in the following way:

$$[\![\mathcal{T}(\mathit{Observer}), P_{StateConsistent}]\!] = \{t \in [\![\mathcal{T}(\mathit{Observer})]\!] : P_{StateConsistent}.t\}$$

Properties of interaction scenarios are, if at all, stated informally in pattern descriptions. This example shows how such properties can be formalized straightforwardly, thus reducing the ambiguity arising from incomplete or informal descriptions.

4.4.2 Pipe

As a second example for the benefits gained from the formalization of our graphical notation we recall the pipe architecture of Section 3.4. Taking a closer look at Figure 19 we notice that the reader may read the pipe without the writer ever having sent a write(value) message to the pipe. In fact, nothing is said about the FIFO properties a pipe typically has. It is difficult, if not impossible, to express these properties in our EET notation. Our trace semantics, on the other hand, allows us to formulate predicates over traces. Again, we first state the <EETL> expressions corresponding to the EETs of Figure 16 to Figure 19:

$\mathcal{T}(\mathit{Write})$ = **(**Writer**,** Pipe**,** write**, (**value**))**

$\mathcal{T}(\mathit{Read})$ = **(**Reader**,** Pipe**,** read**); (**Pipe**,** Reader**,** data**, (**value**))**

$\mathcal{T}(\mathit{WriterCloses})$ = **(**Writer**,** Pipe**,** close**);**
(**(**Reader**,** Pipe**,** read**); (**Pipe**,** Reader**,** data**, (**value**))**)***;**
(**(**(Reader**,** Pipe**,** read**); (**Pipe**,** Reader**,** eof_msg**)**) **| empty);**
(Reader**,** Pipe**,** close**)**

$\mathcal{T}(\mathit{ReaderCloses})$ = **(**Reader**,** Pipe**,** close**); (**Writer, Pipe**,** write**, (**value**))*;**
(Writer**,** Pipe**,** close**)**

$\mathcal{T}(\mathit{Pipe})$ = ($\mathcal{T}(\mathit{Write})$ **|** $\mathcal{T}(\mathit{Read})$)***;** ($\mathcal{T}(\mathit{WriterCloses})$ **|** $\mathcal{T}(\mathit{ReaderCloses})$)

Now we are ready to restrict the set of traces contained in $[\![\mathcal{T}(\mathit{Pipe})]\!]$ to those

- that exhibit FIFO behavior
- where an eof_msg message is transmitted to the reader only after all data messages have been delivered.

An appropriate predicate for the FIFO property is as follows ($t \in AMsg^*$):

$$P_{FIFO}.t \equiv (\forall t': \; t' \in AMsg^* : t' \sqsubseteq t \Rightarrow paramValues^{\omega}.(M_{data} © t') \sqsubseteq paramValues^{\omega}.(M_{write} © t'))$$

where the sets M_{data} and M_{write} denote the sets of all actual read and write messages, respectively:

$M_{data} = \{m \in AMsg : msgName.m = data\}$,

$M_{write} = \{m \in AMsg : msgName.m = write\}$

P_{EOF} ensures that the reader has read all messages from the pipe before an eof_msg message is transmitted ($t \in AMsg^*$):

$$P_{EOF}.t \equiv \quad (\forall t': \quad t' \in AMsg^* : \; t' \circ \langle(Pipe, Reader, eof\_msg)\rangle \sqsubseteq t \Rightarrow \#(M_{data} © t') = \#(M_{write} © t'))$$

The resulting predicate for PIPE is therefore ($t \in AMsg^*$):

$$P_{PIPE}.t \equiv P_{FIFO}.t \wedge P_{EOF}.t$$

This predicate is true, iff trace t has FIFO and EOF property. Thus, the set of all desired traces in our pipe example is

$$[\![\mathcal{T}(\mathit{Pipe}), P_{PIPE}]\!] = \{t \in [\![\mathcal{T}(\mathit{Pipe})]\!] : P_{PIPE}.t\}$$

5 Comparing EETs and WRIGHT Connectors

In this section, we compare the Extended Event Traces as introduced in the previous sections with connector specifications in WRIGHT. WRIGHT is another specification technique for the interaction of architectural components, based on CSP. It is described by Allen and Garlan in [AG94], and allows the software architect to specify both components and connectors separately. Connectors define the roles of interacting components and their coordination in a certain kind of interaction. Because connectors are a “first class” element in this approach, we can reason about component interactions independently, without taking the behavior of components into account.

This section first briefly introduces the notation of WRIGHT. Then, we compare WRIGHT connector specifications to interaction specifications using EETs by means of the Client/Server and pipe architectures.

5.1 Connector Specification in WRIGHT

In [AG94], Allen and Garlan introduce connectors as “explicit semantic entities” for the description of interactions between architectural components. In their approach the architecture of a system is described by a WRIGHT specification that consists of component and connector declarations, a list of component and connector instance definitions, and a list of attachment declarations. A component declaration describes the behavior of an architectural component and the interface for linking its instances with connector instances. A connector declaration defines a certain kind of interaction between components. We will describe connectors in greater detail, below. The list of instances defines the actual entities (components and connectors) that form the system. The list of attachment declarations defines how components are linked via connectors. Without going into syntactic details, we illustrate this notation by the following example, which is extracted from [AG94]:

```
System SimpleExample
    Component Server
         Port provide [provide protocol]
         Spec [Server specification]
    Component Client
         Port request [request protocol]
         Spec [Client specification]
    Connector C-S-connector
         Role client [client protocol]
         Role server [server protocol]
         Glue [glue protocol]
Instances
    s: Server
    c: Client
    cs: C-S-connector
Attachments
    s.provide as cs.server
    c.request as cs.client
end SimpleExample.
```

Figure 26 : WRIGHT Specification of a Client/Server System

Here, two component types (Client and Server) and one connector type (C-S-connector) are declared. Then, an instance of each of the respective types is defined (named s, c and cs). Finally, in the “Attachments” section, s and c are linked via connector cs. Thus, SimpleExample describes a simple Client/Server architecture.

Now we describe the notion of connector in more detail. As stated above, connectors define how architectural components interact. In a WRIGHT specification components do not interact directly but are linked via instances of connector types. Because connector types are declared separately from component types, interaction protocols can be reused in the description of an architecture. We may, for instance, add several server and client instance pairs to the above example. The link between client and server instance within any of these pairs could be established by a separate instance of type C-S-connector. A connector declaration consists of a list of role declarations and a glue declaration. Informally, a role declaration describes how a single participant in an interaction is expected to perform. The glue declaration is used to coordinate the roles of a connector. In [AG94] Allen and Garlan use CSP-like notation to describe role and glue declarations. Furthermore, they define the semantics of WRIGHT specifications by means of CSP processes. Thus, formal reasoning about component interactions is possible. In particular, one can determine whether port declarations (in component specifications) are compatible with role declarations (in connector specifications), such that we are able to determine whether an instance of a given component type can participate in an interaction described by a given connector type. Because a formal semantics for role and glue specifications is defined, certain properties of connectors (like, for instance, deadlocks within connector declarations) can be formulated and (automatically) checked.

Again, we illustrate the connector concept informally, by means of an example, taken from [AG94].

```
connector C-S-connector =
        role Client = (request!x → result?y → Client) ⊓ √
        role Server = (invoke?x → return!y → Server) [] √
        glue = (Client.request?x → Server.invoke!x → Server.return?y
               → Client.result!y → glue) [] √
```

Figure 27 : WRIGHT Specification of a Client/Server Connector

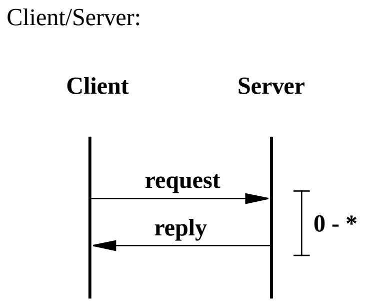
Here, two roles (Client and Server) and their coordination (glue) are declared. The client role consists of requesting a service and receiving the corresponding result. The role of the server is to wait for a service request and to return the result of the service invocation. The coordinating glue defines the order in which these events occur. Note that both the roles and the glue are modelled as CSP processes. In the semantics of WRIGHT the coordination of the role processes is achieved by composing them with the glue process by the ||-operator of CSP.

For a more detailed treatment of the syntax and semantics of connector declarations we refer the reader to [AG94].

5.2 Comparing EETs and WRIGHT Connectors by Examples

In the following, we compare EETs and WRIGHT connector specifications as notations for the specification of component interactions in software architecture. For this purpose, we first consider the Client/Server example described above and in Section 3.1, and then turn our attention to the pipe example of Section 3.4.

In Section 3.1 we have already described a Client/Server system with one client and one server. Figure 28 depicts the EET together with its textual representation and its semantics.



𝒯(*Client/Server*) = ((Client, Server, request); (Server, Client, reply))*

The semantics of this expression is the following trace set:

⟦𝒯(*Client/Server*)⟧ =
{ ε,
< (Client, Server, request), (Server, Client, reply) >,
< (Client, Server, request), (Server, Client, reply),
(Client, Server, request), (Server, Client, reply) >,
< (Client, Server, request); (Server, Client, reply),
(Client, Server, request); (Server, Client, reply),
(Client, Server, request); (Server, Client, reply) >,
...
}

Figure 28 : EET for the Client/Server Interaction Architecture and its Semantics

When comparing this description with the WRIGHT specification of Figure 27 the following differences strike out:

In the WRIGHT specification the interaction protocol is described by a connector that declares two roles: Client and Server. The corresponding glue defines the order in which the interaction between Client and Server takes place. The semantics of the connector description is obtained by composing the three processes (Client, Server and glue) by the ||-operator of CSP. The behavior description is spread over different processes. This leads to redundancies in the system specification, which induces the risk of inconsistencies. The EET describing the same system behavior consists only of two interacting components. There is nothing that explicitly represents the glue and thus, redundancies are avoided.

The case of success denoted by the parallel √-action in every role and the glue is simply given in the EET by the end of the trace at the bottom of the EET. Recursion in CSP is modelled by the repetition operator in the EET.

The WRIGHT specification contains four events:

- the client sends `request!x`, which is received by the glue (`Client.request?x`)
- the glue sends `Server.invoke!x`, which is received by the server (`invoke?x`)
- the server sends `return!y`, which is received by the glue (`Server.return?y`)
- the glue sends `Client.result!y`, which is received by the client (`result?y`)

These four events can be collapsed into two events of the EET, because the EET does not have any glue component. Each message is sent directly from the client to the server, no “intermediate” events, modeling message transportation, are necessary. We expect that, when coding the Client/Server system, the glue will not be implemented by a separate software component.

Further differences between WRIGHT and EET specifications become transparent when considering a more complex example. Therefore, we use the pipe architecture of Section 3.4 and compare it with a similar WRIGHT specification taken from [AG94] (see Figure 29).

```
connector Pipe =
        role Writer = write!x → Writer ⊓ close → √
        role Reader = let ExitOnly = close → √
                      in let DoRead = (read?x → Reader [] read-eof →
                      ExitOnly)
                      in DoRead ⊓ ExitOnly
        glue = let ReadOnly = Reader.read!y → ReadOnly
                            [] Reader.read-eof → Reader.close → √
                            [] Reader.close → √
               in let WriteOnly = Writer.write?x → WriteOnly
                                [] Writer.close → √
               in Writer.write?x → glue [] Reader.read!y → glue
                  [] Writer.close → ReadOnly [] Reader.close → WriteOnly
        spec ∀ Reader.read_i!y • ∃ Writer.write_j?x • i = j ∧ x = y
             ∧ Reader.read-eof
                   ⇒ (Writer.close ∧ #Reader.read = #Writer.write)
```

Figure 29 : WRIGHT Specification of a Pipe Connector (taken from [AG94])

In the WRIGHT specification, the pipe is modelled as a connector. The message exchange of the two roles Writer and Reader (representing the communication restrictions of components, that interact via connector pipe) is synchronized by the glue specification. The description of the connector behavior is achieved by composing the three processes (Writer, Reader and glue) by the ||-operator of CSP. In the EET in Figure 19, the roles, as described by the connector specification of Figure 29, correspond to components and the pipe is modelled as an additional component. Therefore, in this example, no events of the WRIGHT specification are collapsed in the EET. Both of the modelling techniques require an additional predicate over the possible set of traces to describe advanced properties of the pipe, like, for instance, FIFO behavior.

However, this example shows that connector specifications in WRIGHT for nontrivial examples can be fairly hard to read and to understand. This has (at least) two reasons. First, different choice operators ($\sqcap$ denotes internal and $[]$ denotes external choice) are used here for an implicit coordination of the roles. Second, the various execution paths are hard to follow in the textual protocol representation, because a substantial amount of redundancy in system specification is introduced. This is due to the fact that in a WRIGHT connector specification interaction protocols are described in (at least) two locations: in the role and glue part of a connector declaration. Furthermore, a corresponding interaction sequence has to be specified in the port declaration of a component that is designed to interact via the given connector. These redundancies lead to a substantial amount of proof obligations to ensure the consistency of a specification. To decide which action can take place next, it is not sufficient only to look at one process description but it is necessary to look at the parallel composition of the role and glue processes.

On the other hand, the two-dimensional representation of EETs allows us to follow the different execution paths visually. In the example above, it can easily be seen that after an initial phase of unconstrained write and read operations either the reader or the writer can close its connection.

The comparisons above show that EETs are an intuitive and easy to read notation for component interaction with less redundancies than WRIGHT connector specifications. However, WRIGHT deals with both connector and component specifications while EETs focus on component interactions. The interplay between EETs and component specifications has yet to be investigated. For the description of more advanced properties of an interaction architecture both description techniques require additional predicates based on a formal semantics.

6 Conclusion and Further Work

In recent years, caused by the growth of both the size and the complexity of software, the importance of software architecture has increased. Two important parts of an architectural description of a software system are the specification of the participating components and their interaction. Therefore, we enhanced the EETs of [SHB96] to provide an adequate description technique for component interactions with an underlying denotational semantics.

As shown by the examples in Section 3, EETs provide a very intuitive graphical notation for the description of component interaction. Boxes allow us to structure the EETs and thus, help to increase the readability and to reuse interaction patterns by adapting them to a new context. The repetition indicator and the choice operator, as well as the interleaving operator are an additional means to structure complex EETs. Furthermore, these powerful operators enable the developer to specify all communication histories of an interaction protocol.

We have assigned a formal semantics to EETs, which is based on the mathematical concept of streams over communication actions. Thus, the user can specify more sophisticated interaction properties by providing additional predicates over the trace sets corresponding to an EET directly.

EETs avoid the redundancies of connector descriptions in WRIGHT, because the roles of the components and the coordination of the interaction is depicted in a single diagram. In our opinion the usage of CSP-like notation limits the user spectrum of the theory significantly. Practitioners in the field of software architecture and software engineering are seldom trained in the application of process algebras. On the other hand, the notation of EETs is similar to that of Message Sequence Charts, which are well known and extensively used in various modelling techniques. The use of boxes with substitution in EETs allows us to reuse interaction protocols in various contexts, similar to the reuse of connector types.

There are three main areas for further work: First, the language of the EETs has to be evaluated and additional useful operators should be investigated. For instance, we are experimenting with notations for the specification of broadcast messages to groups of components. Examples for additional desirable operators are component creation and deletion. Furthermore, we evaluate the usefulness of other powerful operators, e.g. a “self-interleaving” operator whose semantics is a combination of repetition and interleaving. Concerning the underlying semantics, we examine the benefits of using infinite traces, which would allow us to express liveness properties.

Second, we will investigate how EET descriptions may be integrated into the whole development process and for what purposes they might be used besides the pure documentation of an interaction. A crucial aspect is checking of compatibility of a given component with an interaction architecture specified by EETs.

Finally, methodological questions have to be examined, e.g. which interaction properties should be specified with EETs and which properties should be specified by predicates.

Acknowledgements

The authors are grateful to Markus Kaltenbach, Barbara Paech, Bernhard Rumpe, Bernhard Schätz, and Marc Sihling for stimulating discussions about draft versions of this work.

References

[AG94] R. Allen, D. Garlan: Formal Connectors, Technical Report CMU-CS-94-115, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, 1994

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: A System of Pat terns. Pattern-Oriented Software Architecture. Wiley, Sussex, 1996

[Boo94] G. Booch: Object-Oriented Analysis and Design with Applications. 2nd ed. Addi-son-Wesley, CA, 1994

[BRJ96] G. Booch, J. Rumbaugh, I. Jacobson: The Unified Modeling Language for Object-Oriented Development, Version 0.9, 1996

[Fac95] C. Facchi: Methodik zur formalen Spezifikation des ISO/OSI Schichtenmodells. PhD-Thesis. Technische Universität München, 1995

[GHJ+95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, CA, 1995

[HHK+96] C. Hofmann, E. Horn, W. Keller, K. Renzel, M. Schmidt: The Field of Software Architecture. Technical Report TUM - I9641, Technische Universität München, 1996

[ISO94] ISO: Final DIS text of ISO/IEC 10731, information technology - Open Systems Interconnection - conventions for the definition of OSI services. Technical Report ISO/IEC JTC 1/SC 21 N 8604, ISO, 1994

[ITU94] International Telecommunication Union: Message Sequence Charts. ITU-T Recommendation Z.120. Geneva, 1994

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorenson: Object-Oriented Modeling and Design. Prentice Hall, 1991

[SHB96] B. Schätz, H. Hußmann, M. Broy: Graphical Development of Consistent System Specifications. In: J. Woodcock, M.-C. Gaudel, eds.: FME'96: Industrial Benefit and Advances in Formal Methods. Springer, LNCS 1051, 1996