Using Extended Event Traces to Describe Communication in Software Architectures[1]

Manfred Broy, Christoph Hofmann, Ingolf Krüger, Monika Schmidt
Institut für Informatik
Technische Universität München
D-80290 München, Germany
{broy, hofmannc, kruegeri, schmidtm}@informatik.tu-muenchen.de

Abstract

A crucial aspect of the architecture of a software system is its decomposition into components and the specification of component interactions. In this report we use a variant of Extended Event Traces [15] 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. Our notation contains operators yielding not only exemplary but complete behavior specifications. Extended Event Traces have a clear semantics that is based on sets of traces. We present several application examples that demonstrate the practical use of our notation.

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 [16][11]. Although no satisfactory formal definition of the term software architecture (or architecture, for short) exists to date, most researchers in the field agree on the following architectural constituents: components and their relationships.

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.

The research for a description technique for communication in software architectures was motivated by our project ENTSTAND (in cooperation with our industrial partner sd&m[2]), where architectures of business information systems and frameworks are specified and investigated. An adequate notation for component interaction in such architectures should be easily applicable, understandable, and based on a formal semantics. Furthermore, not only exemplary but complete communication histories should be describable.

In this report, we use a graphical notation for the description of component interaction in software architectures. Our notation is based on Extended Event Traces (EETs, [15]), which are similar to Message Sequence Charts [12]. We enhance this notation by additional operators that allow us to describe interaction architectures succinctly. The EETs can be translated to sets of traces of system events in a straightforward manner. By formulating predicates over these traces special properties can be expressed. Our EET notation has a denotational semantics based on traces of system events. In this report, we omit the presentation of the formal semantics and refer the interested reader to [3].

Other specification techniques for component interaction in software architectures, which also have a formal

[1]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'.

[2]software design & management (sd&m) GmbH & Co KG, München, Germany

semantics, often provide a less intuitive notation. For instance, in WRIGHT [1], CSP processes are used both as the notation for interaction specifications and their semantics. In our opinion graphical description techniques are better suited for application in industry because they require less training than process algebras.

The remainder of this report 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 and indicate how additional properties can be specified by predicates over traces. Finally, Section 4 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 [15] as the basis for our notation. In Section 2.2, we illustrate the syntax and its informal semantics by means of examples. Section 2.3 contains an outline of the relationship between the informal and formal semantics.

2.1 Extended event traces

Event traces emerged in the field of telecommunications (see, for instance, [12]) and are now extensively used in various modeling techniques, such as object-oriented analysis and design methods [5][6] and architecture descriptions [9][4] 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 a lot of training. Often, however, their semantics is not formally defined, which leads to ambiguities in specifications.

In [15], 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 offer 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 [15] 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 [15] 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 [3].

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 different axes labellings and, therefore, 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. 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).

EET$_1$:

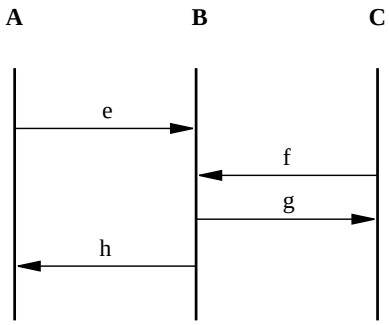


Figure 1. Simple EET

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.

EET$_2$:

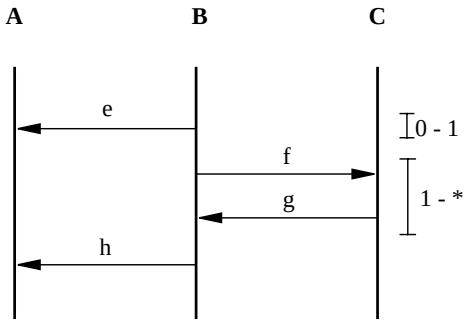
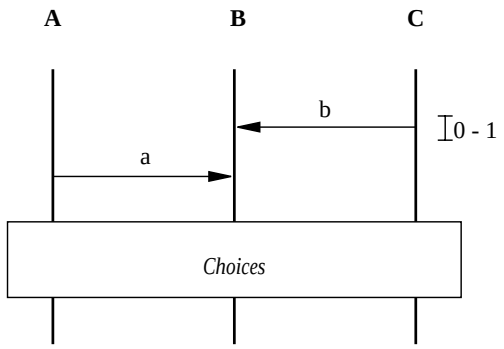


Figure 2. EET with optional and repeated parts

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.

EET$_3$:



Choices = {EET$_1$, EET$_2$}

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 [15] explicitly allows recursive diagrams.

Rec:

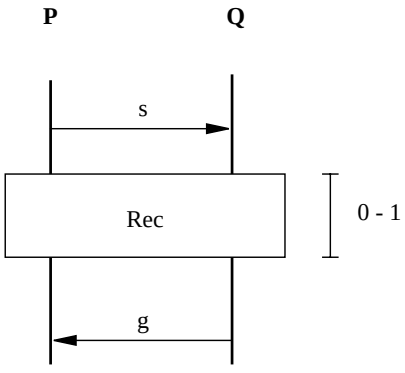


Figure 4. Recursive EET definition

If we consider EETs as graphical representations of formal languages the definition given in [15] 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 3 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.

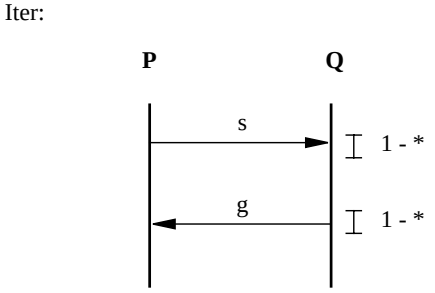


Figure 5. Nonrecursive EET definition

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) 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.

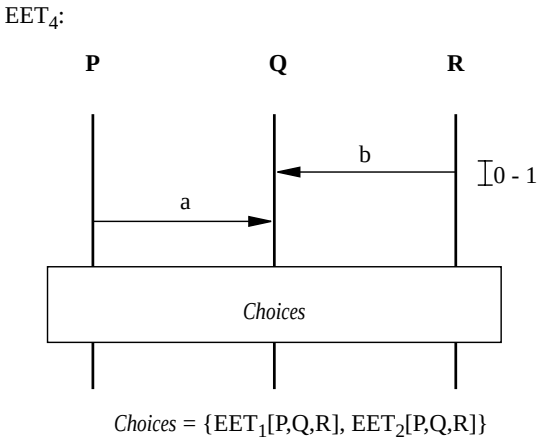


Figure 6. Adaption of context

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 [15] 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 [15].

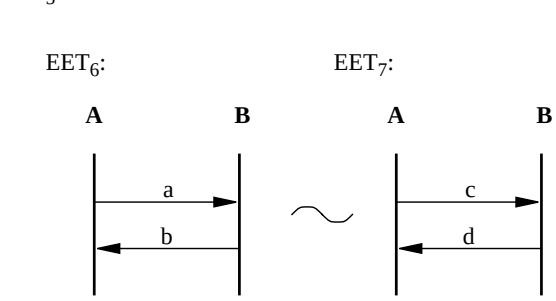


Figure 7. Interleaving operator

2.3 Trace semantics

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 (cf. [3]).

Here, a trace is a finite sequence of events where each event is denoted in the following form

$(S, R, mn, (v_0, ..., v_{n-1}))$

S represents the component that sends the message with name mn, and the component represented by R receives it. The message may contain a list of parameters $(v_0, ..., v_{n-1})$. An empty parameter list may be omitted. Note that concrete values have to be substituted for the formal parameters of the messages when building the trace.

3 EETs for example architectures

In this section, we give four examples of interaction architecture specifications using EETs (some of them are inspired by [1]). They demonstrate the practical use of all operators introduced in Section 2.2. Although the examples presented here are relatively small, EETs are applicable to more sophisticated interaction architectures as well (cf. Section 4).

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.

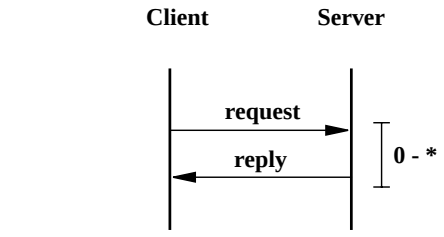


Figure 8. EET for the Client/Server interaction architecture

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.

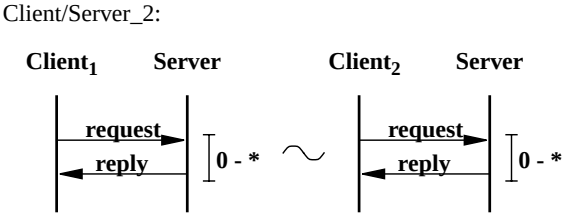


Figure 9. EET for the interaction of two clients with a single server

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.

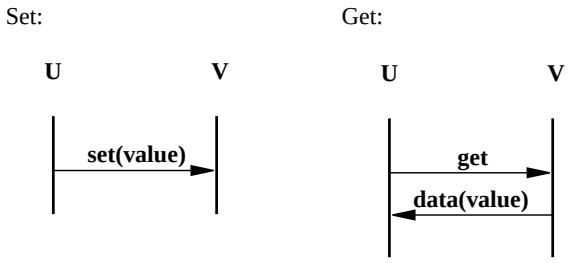
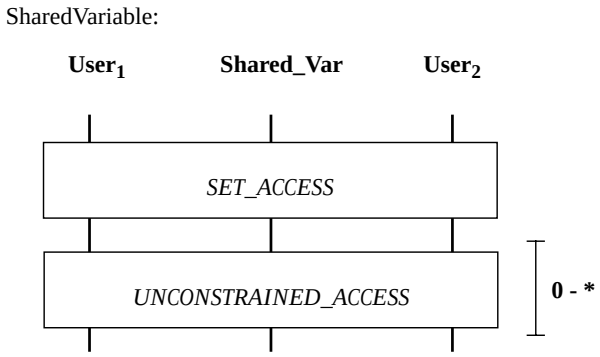


Figure 10. Variable access

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.



SET_ACCESS =
{Set [$User_1$, Shared_Var],
Set [$User_2$, Shared_Var] }

UNCONSTRAINED_ACCESS =
{Set [$User_1$, Shared_Var],
Set [$User_2$, Shared_Var],
Get [$User_1$, Shared_Var],
Get [$User_2$, 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 [9] [4] 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 [9]. 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 [4], 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 [14], is given in Figure 12.

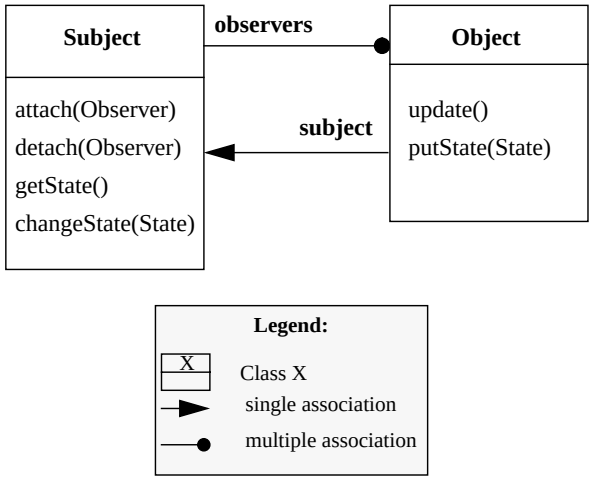


Figure 12. Simplified structure of the observer pattern

In [9], 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.

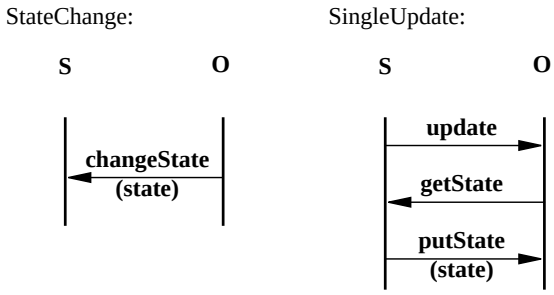


Figure 13. State change of the subject and state transmission to a single observer

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).

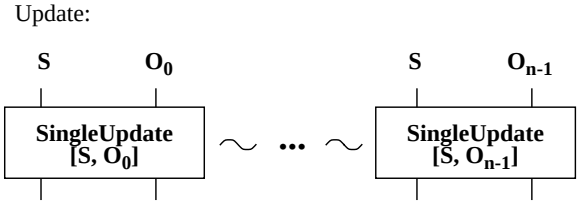
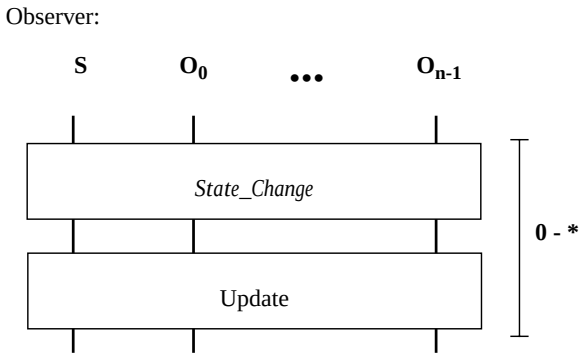


Figure 14. Updating the state information of all registered observers O_0, ..., O_{n-1} with subject S

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).



State_Change = {StateChange[S, O_0], ..., StateChange[S, O_{n-1}]}

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 [9] and [4].

Properties of interaction scenarios are, if at all, stated informally in pattern descriptions. An example of such a property is that within one update cycle every Observer receives the same state value. In [3] we show how such properties can be formalized straightforwardly, thus reducing the ambiguity arising from incomplete or informal descriptions. The following section contains an example of such a formalized property.

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.

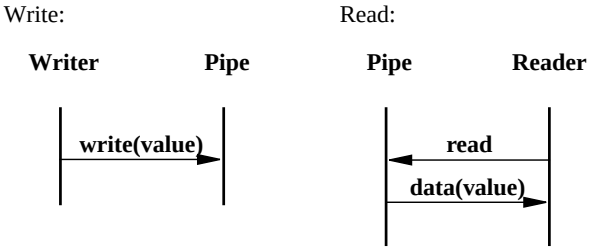


Figure 16. Read and write scenario

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.

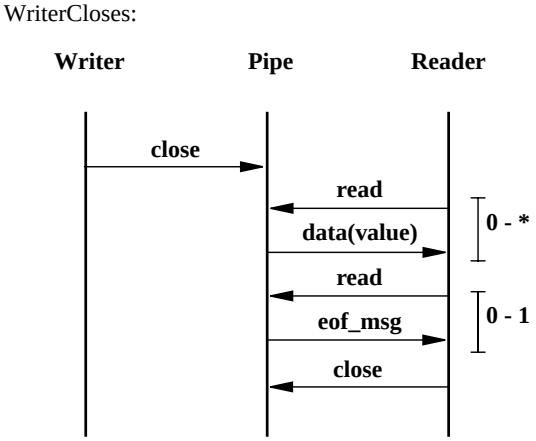


Figure 17. “Writer closes” scenario

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).

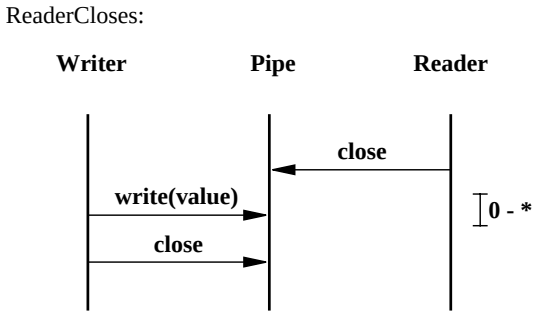


Figure 18. “Reader closes” scenario

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.

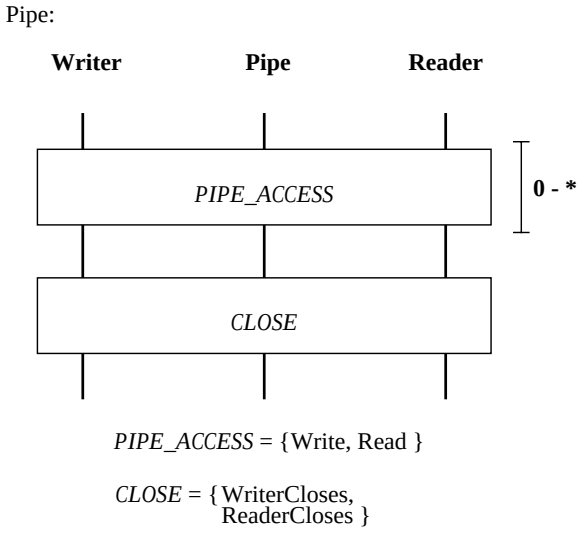


Figure 19. Pipe interaction architecture

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 *CLOSE*).

Note that we cannot specify the FIFO property, which a pipe typically has, in our graphical notation. Our trace semantics, however, 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). To demonstrate this approach we show how to specify the following properties of our pipe architecture by means of predicates:

- the pipe has FIFO behavior
- the pipe transmits an eof_msg message to the reader only after all data messages have been delivered.

To formulate the corresponding predicates we introduce some notation. By Msg we designate the set of all events. We represent traces of events as finite streams over set Msg. A stream over a set M is a finite or infinite sequence of elements from M. By M^* we denote the set of finite sequences of elements from 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:

$$\begin{array}{ll} \& & : M \times M^* \to M^* \\ \# & : M^* \to \mathbb{N} \\ <.> & : M \to M^* \\ \circ & : M^* \times M^* \to M^* \\ \copyright & : \wp(M) \times M^* \to M^* \\ . \sqsubseteq . & : M^\omega \times M^\omega \to \mathbb{B} \end{array}$$

By $\mathbb{B} = \{true, false\}$ and $\mathbb{N}$ we denote the set of Boolean truth values and the set of natural numbers, respectively. For $m \in M$, $s, t \in M^*$ and $N \subseteq M$ the purpose of these operators can be described as follows: m&s yields the stream whose first element is m and then continues as s. #s determines the length of (i.e. the number of elements in) s. The term <m> denotes the stream consisting of only the element m. $s \circ t$ yields the stream obtained by prepending stream s to stream t. 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 and their extension to infinite streams cf., for instance, [7][8].

Furthermore, we define the following function and abbreviations (MN and VAL denote the set of message names and the set of parameter values, respectively):

$$\begin{array}{l} msgName : Msg \to MN \\ M_{data} = \{m \in Msg : msgName.m = data\} \\ M_{write} = \{m \in Msg : msgName.m = write\} \\ paramValues : Msg^* \to VAL^* \end{array}$$

where msgName.m denotes the name of message m. Hence, M_{data} and M_{write} denote the subsets of Msg containing all events labelled with data and write, respectively. For any $t \in Msg^*$, paramValues.t yields the stream consisting of the parameter tuples of the messages in t.

Now we are ready to formulate the properties mentioned above by means of predicates. An appropriate predicate for the FIFO property is as follows ($t \in Msg^*$):

$$\begin{array}{l} P_{FIFO}.t \equiv \\ (\forall\, t' : t' \in Msg^* : \\ \qquad t' \sqsubseteq t \Rightarrow \quad paramValues.(M_{data} \copyright t') \\ \qquad\qquad\qquad \sqsubseteq paramValues.(M_{write} \copyright t')) \end{array}$$

Intuitively, $P_{FIFO}.t$ states that in any prefix t' of t the substream of t' containing only the parameter values of data messages is a prefix of the substream of t' that contains only the parameter values of write messages. Hence, there are at most as many data as write messages in t', and the parameter values of corresponding write and data messages are the same.

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

$$\begin{array}{l} P_{EOF}.t \equiv (\forall\, t' : t' \in Msg^* : \\ \qquad\qquad t' \circ <(Pipe, Reader, eof\_msg)> \sqsubseteq t \\ \qquad\qquad \Rightarrow \#(M_{data} \copyright t') = \#(M_{write} \copyright t')) \end{array}$$

The resulting predicate that describes the desired interaction properties of the pipe architecture is therefore ($t \in Msg^*$):

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

Thus, the set of all desired traces in our pipe example is

$$\{t \in [\![Pipe]\!] : P_{PIPE}.t\},$$

where $[\![Pipe]\!]$ denotes the set of traces as described by EET *Pipe* of Figure 19. For the details of our semantic treatment of EETs we refer the reader to [3].

4 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 [15] 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 inter-

action 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. Additionally, the user can specify more sophisticated interaction properties by providing predicates over the trace sets corresponding to an EET directly.

The notation of EETs is similar to that of Message Sequence Charts, which are well known and extensively used in various modelling techniques. Similar notations are also used to describe communications in design patterns. The granularity of design patterns varies from particular design problems [4][9] to designs of complex system architectures [13]. To date, all pattern descriptions have in common, that component interactions within patterns are described by exemplary scenarios. EETs, on the other hand, can be used to describe complete interaction behavior and, thus, can reduce the ambiguity of pattern descriptions (see [3]). The reuse of design ideas is supported by the EET notation, because boxes with substitution in EETs allow us to reuse interaction protocols in various contexts.

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 and operators for component creation and deletion. Such operators are especially useful when applying the notation to complex architectures, such as the relational database access layer described in [13]. Second, we will investigate how EET descriptions may be integrated into the whole development process. The formal semantics presented in [3] allows us to relate EETs with other description techniques (such as State Transition Diagrams [10][2]), thus supporting multiple views on the system under development consistently. A crucial aspect in this context is ensuring 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, and to the anonymous referees for their comments.

References

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

[2] M. Broy. The Specification of System Components by State Transition Diagrams, Technical Report TUM-I9729, Technische Universität München, 1997

[3] M. Broy, C. Hofmann, I. Krüger, M. Schmidt. A Graphical Description Technique for Communication in Software Architectures. Technical Report TUM-I9705, Technische Universität München, 1997

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

[5] G. Booch. *Object-Oriented Analysis and Design with Applications.* 2nd ed. Addison-Wesley, CA, 1994

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

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

[8] C. Facchi. Formal Semantics of Time Sequence Diagrams. Technical Report TUM-I9540, Technische Universität München, 1995

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

[10] R. Grosu, C. Klein, B. Rumpe, M. Broy. State Transition Diagrams, Technical Report TUM-I9630, Technische Universität München, 1996

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

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

[13] W. Keller, J. Coldewey. Relational Database Access Layers - A Pattern Language, accepted for Pattern Languages of Program Design, Volume III, Addison-Wesley, 1997, to appear

[14] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, 1991

[15] 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

[16] M. Shaw, D. Garlan. *Software Architecture — Perspectives on an Emerging Discipline*, Prentice Hall, 1996