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 Framework for Interaction Description with Roles

Barbara Paech

TUM-I9731
Juni 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

TUM-INFO-06-I9731-150/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

A Framework for Interaction Description with Roles *

Barbara Paech
Institut für Informatik, Technische Universität München
D-80333 München, Germany
email: paech@informatik.tu-muenchen.de

Abstract

We propose a framework for interaction description based on the paradigm of roles to be used in software development. It includes processes to formulate global interaction requirements abstracting from configuration and interaction details and event traces to analyze global interaction properties like deadlock-freedom. The main constituent of the framework are role descriptions which are structured into state space, services, configuration and interaction. They capture the component behaviour in a particular context. This allows for the description of a wide range of software architectures and designs. Our approach ties together work in the area of object-oriented programming languages and software architecture.

Keywords:
Software Architecture, Object-Orientation, Software Development Methodology, Interaction Description, Roles, Processes, Event Traces

*This paper originated in the SysLab project, which is supported by the DFG under the Leibnizpreis and by Siemens-Nixdorf. Its final version was completed while the author was supported by the Bayerische Forschungsstiftung within the ForSoft project.

Contents

1 Introduction

The engineering of complex software systems has made apparent the need for *interaction descriptions* on different levels of abstraction. In the realm of object-oriented programming languages several proposals have been made for specifying the collective behaviour of object systems (e.g. [HHG90, KM96]). Interaction specification is also an important topic in the emergent field of software architecture (e.g. [PW92, AG94, MDEK95, SG96]). In addition to specification, the latter is also concerned with analysis of interaction descriptions.

The aim of this paper is to tie together these efforts within a framework of interaction description ranging from the specification of interaction requirements through architectures and designs with explicit interaction specification to the analysis of global interaction properties. While our approach is not yet complete, we want to outline here the major description techniques and the problems which can be tackled within the framework.

In the first part of the paper we describe the ROLE language for specifying architectures and designs. The distinction between architecture and design is taken from [PW92]: architectures are concerned with the selection of components, their interactions and constraints on the interactions, while designs are concerned with the details of the components. We use the same description technique for both levels, but allow for different levels of abstraction covering this distinction.

We use *roles* for specifying components *in a particular context*. Complete component behaviour is obtained through composition of its roles. To support a clear separation between architectural concerns and computational concerns and the localization of interaction information (required for interaction descriptions in [SG96]) a role description is structured into state space, functionality, configuration and interaction. Each element can be specified on different levels of abstraction.

In the second part of the paper we discuss the analysis of *global interaction properties* and the relationship of ROLE specifications to *global interaction requirements*. The distinction between property and requirement is made apparent through different description techniques: we use *event traces* (as used in [Jac92, BRJ96, IT96, BHKS97]) to describe global interaction sequences of a given architecture/design and we use *processes* (as used for business process and workflow specification, e.g. [Sch92]) to describe interaction dependencies of roles. The main difference is that the processes abstract from configuration details and interaction control which is explicit in the event traces.

The ROLE language was developed within the SYSLAB project which aims at giving a formal foundation to description techniques and tools used in the software development process [Pae95, Bro95]. The formal semantics is given in terms of a mathematical system model. It can be used for analysis and refinement techniques. In the following, we will not go into details of the formal semantics of the description techniques. However, it has been an important concern in the development of the proposed framework.

Altogether, the paper is structured as follows: In the next section we introduce the basic concepts of ROLE. Then we give examples for the more advanced features like dynamic configuration and role refinement. In the fourth section, we show how to analyze global properties of ROLE architectures and designs and we discuss the transition from interaction requirements to ROLE architectures. Related work is discussed along the way, in particular in the last section, where also future work is sketched.

2 The concept of ROLE

In this section we explain the basic features of the ROLE language for specifying software architectures and designs.

We view a software system as a set of concurrently interacting actors (components). Interaction consists of asynchronous message exchange. Each actor offers a set of services. Service calls are a particular type of messages. Actors can constrain the service call acceptance. Services operate exclusively on the data encapsulated by the actor. A mathematical system model along these lines is given in [KRB96].

The architecture/design of a system is described by roles and actors. Roles are used to decompose the data and services of actors into meaningful units to be used in a particular context. The roles of one actor are activated concurrently.

As a first example consider the ubiquitous *Pipe* component. In figure 1 its role description is given. The `attributes` describe the *state space* of the role. The pipe encapsulates a sequence of some data type *data* (for data type specification the algebraic specification language MiniSpectrum [Het96] is used). The communication `partners` determine the *configuration structure* on a logical level. The pipe communicates with one reader and one writer. The partners are fixed for all interactions of the pipe (in the next section we describe how to handle dynamic configurations). They are named for use in the service description and can be restricted to a particular role. In the case of a pipe any role is allowed as reader and writer.

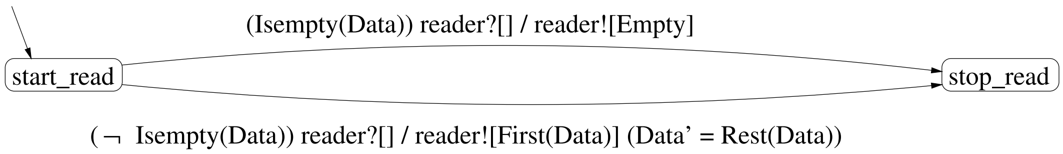
The `services` capture the *functionality* of the role. For each service first the input and output messages are listed. The general syntax is `input message : message type from sender` for input and `output message : message type to receiver` for output. Sender and receiver must be names of communication partners. Each service may use an arbitrary number of messages for communication. The service call message is labelled with `trigger`. If there is no important call parameter, it may be omitted. The service may also have a distinguished `final` output message. With this message the result of service execution is delivered to the callee. Service calls to other roles are of type `signal`. For the pipe the write service and the close services do not deliver output. The read service receives the read request and delivers the data.

The behaviour of a service can be described on different levels of abstraction. No further information need be specified, if only the *configuration structure* is important. The behaviour can be described by pre- and postconditions defining the involved data changes, if only the effects on the state space and the triggering input and final output are relevant. If the full (interaction) behaviour of a service is relevant, it is described by an enhanced form of nondeterminstic input/output automata defined in [GKRB96]. In the example such an automaton is given for the read service.

role pipe = {
attributes Data : sequence of data
partner reader, writer : any
service write = {
    trigger **input** d : data **from** writer
    post Data' = Data $\circ$ d }

service write_close = {
    trigger **from** writer
    post Data' = Data $\circ$ Eof }

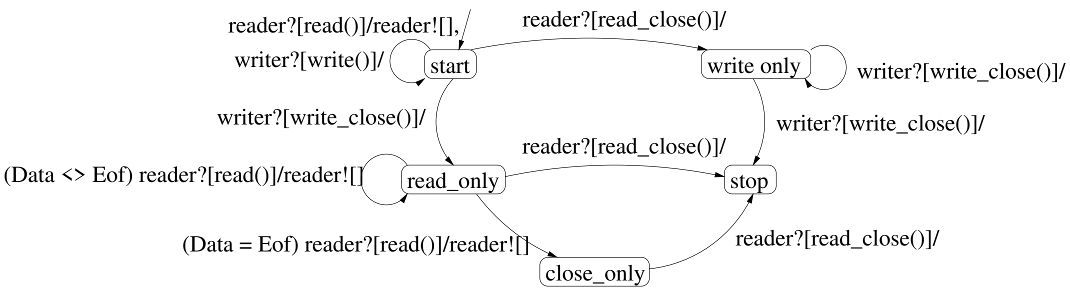
service read = {
    trigger **from** reader
    final **output** d : data $\cup$ Empty **to** reader
    interaction



    }

service read_close = {
    trigger **from** reader }

interaction



}

Figure 1: ROLE description of a pipe

Automata states are labelled with names to distinguish different control states (capturing the sequence of message exchanged so far) [1]. Start states are marked with an incoming arrow. The transitions are labelled with a tupel `(precondition) sender?[input pattern] / receiver![output pattern] (postcondition)`. Either part of the tupel can be omitted. The input pattern specifies a set of input messages, the output pattern a set of output message *sequences*. The preconditions may reference the input message as well as the attributes, the postcondition may reference the input and output messages together with the attributes (similar to TLA [Lam94], we use the bar notation to name the attribute values of the successor state).
In the example, the read service distinguishes two cases: if the data sequence is empty, the Empty message is returned, otherwise the first data element.
The service automata are used to define the full service behaviour at the design level. For architecture description in many cases pre- and postconditions will suffice. There is always one automata included in the role definition (called *interaction automaton*) specifying the major `interaction` of the role. This part describes the behaviour which is activated together with role initialization. Here the dependencies of service acceptance on the control states of the role are defined, as well as the message exchange which does not belong to some service. Note, however, that the state space may only be changed by the services. In the pipe example the interaction automaton is only used to restrict service acceptance depending on the close messages received. For notational convenience input and output patterns may be omitted in the interaction automaton. Service calls are distinguished from normal messages by appending "()" and possibly some input argument, if relevant for the transition.
The information about service call and returned result is duplicated in the interaction automaton and the service automata. While in the interaction automaton it is captured within one transition, the service automaton might specify a complex sequence of interactions in between service acceptance and service completion. Because of exclusive data access of the services no further service calls may be accepted during service execution. This distinguishes services from methods in object-oriented approaches. The complete role behaviour is captured by substituting the service transitions in the interaction automaton with the corresponding service automaton.
The interaction part localizes the information about enabledness of services. It is similar to synchronization constraints in concurrent object-oriented programming languages (e.g. [Nie93]). In contrast to these approaches the interaction automaton may be given on different levels of abstraction. In our framework, nondeterminism corresponds to underspecification. Pre- and postconditions or input and output patterns can be used to refine the behaviour of the automaton.
The easy expression of underspecification is one reason for choosing automata instead of process calculi like CSP [Hoa85]. Also, automata can make the lifecycle of role data explicit. The major advantage of automata is their widespread use and ease of understanding. The latter is especially important for the architectural description which needs

[1] In [GKRB96] this is extended to predicates characterizing data states

to be communicated between software designers and users.

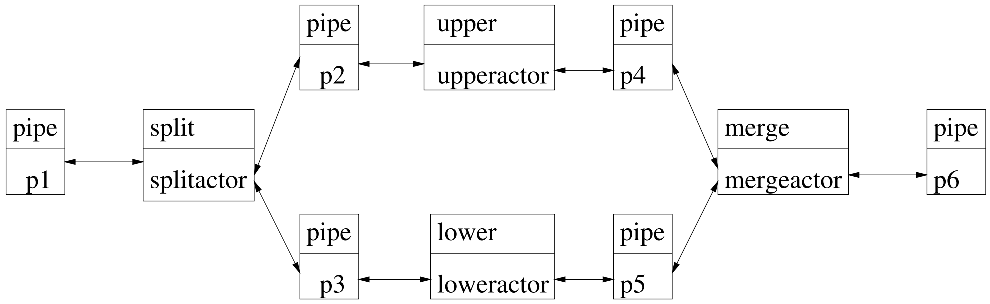


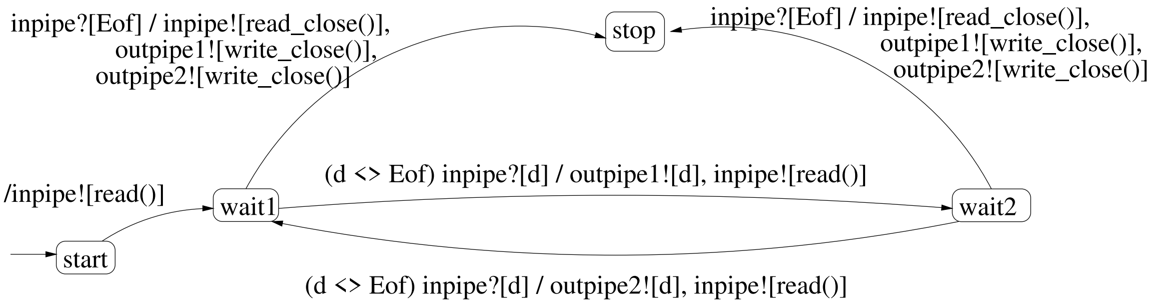
Figure 2: Configuration Structure of CAPITALIZE

A system is specified in terms of roles and actors. Figure 2 gives a graphical description of the configuration structure of the CAPITALIZE system discussed in [SG96]. Boxes correspond to actors and their roles, arrows to partner references. Boxes show the name of the actor in the lower part and the instantiated roles in the upper part. In addition to the pipe role there are four roles corresponding to the filters split, upper, lower and merge, which split an input stream into two streams for the alternate characters, change all characters to upper or lower case and merge the input streams, respectively. The latter roles are instantiated with one actor each, while there are six pipe actors responsible for data transmission between the filter actors.
Figure 3 shows the description for the split role. It requests input from inpipe and distributes it to outpipe1 and outpipe2, alternately until it receives the closing signal. The communication partners are required to be pipes. For reuse it might be desirable to allow more general communication partners, since the filters only use a subset of the functionality of each pipe. This could be captured by role refinement which is discussed in the next section. The filter roles do not offer any service to be called by the environment. Their whole functionality is captured by the interaction part which reads from the input pipes and writes to the output pipes by calling the appropriate services.
The configuration structure can also be given purely on the logical level where only roles are involved. A set of interacting roles is called a *context*[2]. In general each actor will instantiate roles of different *contexts* (this is similar to the use of roles in organizational modelling, e.g. [Yu93]. It does not show up in the simple examples used in the paper.).
Figure 4 gives the textual definitions for actor instantiation. For reasons of space we only give the instantiation for one pipe and one filter actor. It defines for each role of each actor the actors for the communication partners.
At first sight our description of the CAPITALIZE system looks similar to the WRIGHT description given in [SG96]. We, too, have language features for description of components and connectors (namely, roles), and for instantiation and configuration (namely, actors). The interaction is specified in WRIGHT using the glue for connectors, while we use the interaction automata. However, there are important differences.

[2]In the following we use italics to distinguish this meaning of the word context.

```
role split = {
partner inpipe, outpipe1, outpipe2 : pipe
input d : data from inpipe
output d : data to outpipe1, d : data to outpipe2,
    read(), read_close() : signal to inpipe,
    write_close() : signal to outpipe1, write_close() : signal to outpipe2

interaction
```




```
}
```

Figure 3: ROLE description of a split filter

```
actor p1 = {
roles pipe
    partner reader = splitactor
}
...
actor splitactor = {
roles split
    partner inpipe = p1, outpipe1 = p2, outpipe3 = p3
}
....
```

Figure 4: System CAPITALIZE

- First, we do not distinguish between components and connectors. With connectors the interaction description of a set of roles is composed of the role descriptions and the glue. In some cases (e.g. client/server) the glue does not contribute any additional constraint on the role descriptions. It therefore is omitted in our framwork. In other cases (e.g. pipe), the glue describes additional control on the role interactions. This is modelled in our framework as an additional role responsible for facilitating the interaction between partner roles. Often a connector has an important data part (e.g. shared data), which makes it similar to components. For localization of interaction information the distinction of state, functionality, configuration and interaction description within roles seems to be sufficient. Note that, although we dispense with connectors, we introduce the notion of *context* to name a set of interacting roles.
- The interface of roles may be structured through services. This is necessary for roles which encapsulate data. One could argue that for architectural descriptions only the message flow is relevant. In our view, data is very often important to understand the purpose of the role. Here we agree with [PW92] which also argue for the close interdependence of functionality and data.

3 Using ROLE

In this section we want to demonstrate the flexibility of ROLE. In particular we discuss packaging, dynamic configuration, role refinement and representation of architectural styles.

3.1 Packaging

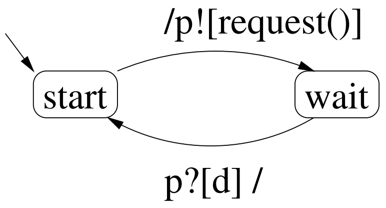
One of the main decision of software architecture is to assign responsibilities for interaction to the components. This is called packaging in [Sha95]. The component taking responsibility is the active part, issuing calls to the environment, while the partners react. These responsibilities are clearly visible in the interaction automata of the roles. In the following, we will discuss several examples demonstrating the different ways of packaging in ROLE.

client/server We start with a simple client/server system. The client is the active part, the server reacts. We just assume one client. The description for several clients is given in the next subsection. In figure 5 the ROLE description of this system is given.

The server interaction automaton shows only reaction to service calls and no outgoing calls. This style is typical for sequential object-oriented systems where every interaction is located in the services.

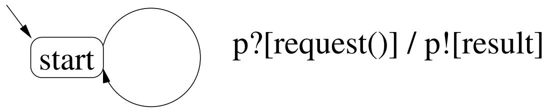
data transmission with active reader Now we look at a simple data transmission between a writer and a reader. We assign the responsibility to the reader, who requests

```
role   static_client = {
partner p : static_server
input d : data from p
output request() : signal to p
interaction
```




```
}
```

```
role   static_server = {
partner p : static_client
service request = {
    trigger from p
    final output result : data to p }
interaction
```




```
    }
}
```

Figure 5: Client/Server Roles with static configuration

data transmission from the writer by calling the read service. Only the reader can shut down the connection. Therefore the writer has a service to receive the closing signal. Figure 6 shows the corresponding ROLE description. The interaction parts of the roles are almost symmetric, since the messages issued on one side are mirrored as accepted messages on the other side. The reader additionally needs a transition to receive the answer to the write call.

The case of an active writer who calls a write service of the reader reverses the responsibility.

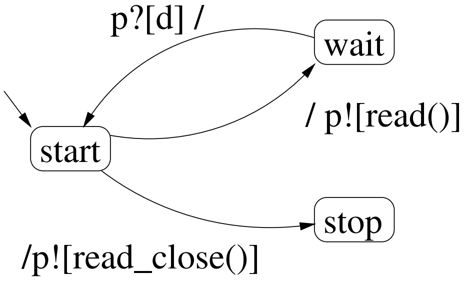
shared responsibility Finally, we look at data transmission where the responsibility for interaction is shared between reader and writer. This means that both roles issue transmission service calls, react to incoming transmission service calls and can shutdown the connection. Figure 7 shows the corresponding ROLE description.

3.2 Dynamic Configuration

So far, only static configurations have been considered. In ROLE this corresponds to partner declaration fixed for all services (as discussed for the CAPITALIZE system). Dynamic configuration is expressed by placing the partner declaration within the service definition. This makes the partner to parameters of the service call.
Figure 8 shows a client/server system with two server-actors and two client-actors. Client1 instantiates two client roles, one for each server. Client2 is only client of server1 and therefore only instantiates one client role.
In our framework roles are instantiated concurrently. Thus the client-actor can issue calls to different servers concurrently. If the interactions with different servers depend on each

role active_reader = {
partner p : passive_writer
input d : data from p
output read(), read_close : signal to p

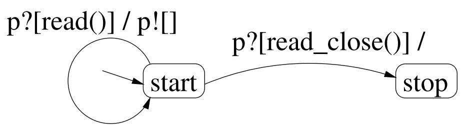
interaction



}

role passive_writer = {
partner p : active_reader
service read = {...}
service read_close = {...}

interaction



}

Figure 6: Data transmission with active reader

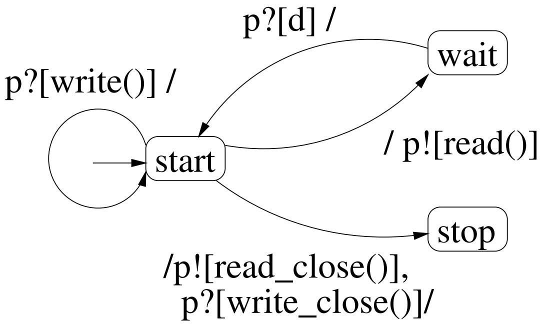
other, the client-actor has to instantiate an additional role coordinating the client-roles (not modelled here).
The server-actors only instantiate one server-role. The server roles fo not fix any communication partner, since the partner of the request service is declared within the service. Therefore the partner binding is done differently for each service call. The transitions of the interaction automaton are labelled with $*$ instead of the partner name.

3.3 Role Refinement

Even more flexibility of ROLE architectures and designs is possible with role refinement. It allows for substitution of partner roles with refined roles exhibiting more complex behaviour. In the realm of object-oriented systems a number of refinement (inheritance) notions have been defined. We adopt our definition given in [PR94]. In that paper we have defined refinement of behaviour automata. These automata are a simpler form of the automata used here, where transitions are only labelled with inputs. In that context refinement allows for addition of new attributes and new services. Also the enabledness of services may be extended, if no additional nondeterminism is introduced, and it may be restricted, if the state space is reduced because of reducing nondeterminism. It is straightforward to adopt this work to outputs, such that also additional calls to new services are allowed.
The main requirement for substitution is that within a *context* the behaviour guaran-

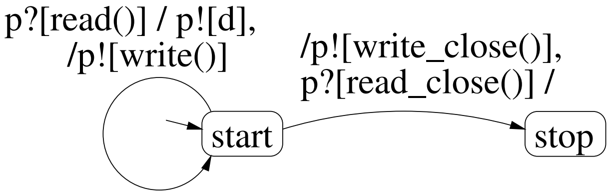
role reader = {
partner p : writer
input d : data from p
output read(), read_close : signal to p
service write = {...}
service write_close = {...}

interaction



}

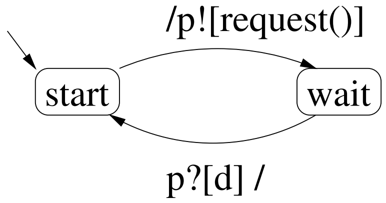
role writer = {
partner p : reader
output write(), write_ close : signal to p
service read = {...}
service read_close = {...}
interaction



}

Figure 7: Data transmission with shared control

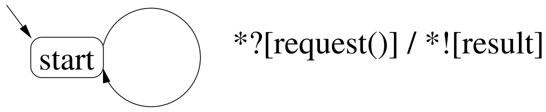
```
role  client = {
partner p : server
input d : data from p
output request() : signal to p
interaction
```




```
}
```

```
role  server = {
service request = {
    partner p : client
    trigger from p
    final output result : data to p
}

interaction
```




```
}
```

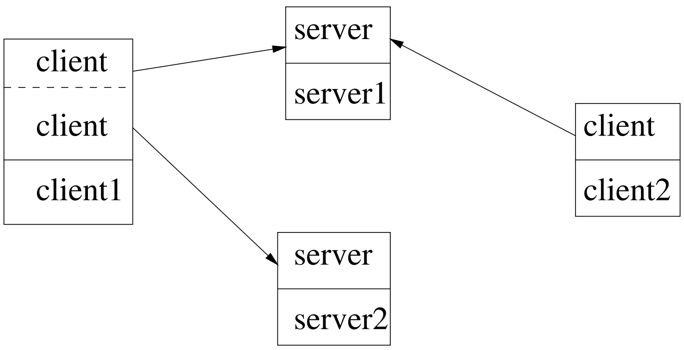


Figure 8: Server with dynamic clients

teed to the environment is preserved. The definition of [PR94] ensures preservation of service enabledness. This does not guarantee that all possible interaction sequences in a given context are preserved, in particular deadlocks may be introduced. Translating the results of [AG94], one can show that deadlock-freeness is preserved, if there is one role coordinating all interactions (the glue) and this role is not refined. It would, of course be desirable to come up with refinement notions preserving at least some class of global interaction properties. As discussed in the next section we are experimenting with event traces to formulate and analyze global properties. We do not go into detail here, but only illustrate the use of role refinement to allow for reuse of roles. As an example consider the substitution of the passive_writer partner (figure 6) by a pipe (figure 1). The start state of the passive_writer is refined to the pipe start state and and all other pipe states refine the stop state of the passive_writer. With this correspondence one can easily see that all transitions of the passive_writer role are preserved.

3.4 Representing Architectural Styles

Different styles are captured by constraints on the use of the ROLE language. As mentioned before, *object-oriented architectures* are easily captured by restricting the interaction automata such that each transition is labelled with an incoming service call (and possible some output). In this style all interactions and functionality are located within the services. Configuration is static or dynamic. Partner declarations global to all services capture the references between classes.

For *data flow architectures* services and attributes are not used. In this style all interactions and functionality are localized in the interaction part. Configuration is static.

Message-passing architectures use components with an explicit state space, which communicate through messages. This corresponds to roles with one distinguished service where all the functionality and interaction is located. The interaction part of such a role is trivial; it consists of an automaton with one transition for acceptance of the event triggering this distinguished service. Configuration may again be static or dynamic.

While the above styles are represented by constraining the basic features of ROLE, *event systems* and *shared data systems* are captured by introducing special roles. Event systems use implicit invocation, where events are broadcast to all components that have registered for that event. Up to now broadcast is not included in ROLE. Therefore, implicit invocation must be modelled by a separate role handling the event broadcast and the registration. It seems straightforward to enhance ROLE with a more flexibel `partner` declaration allowing sets of partners that are managed by the roles or services using some binding mechanism. However, we have not worked out the details yet, in particular one has to incoporate this new feature into the global interaction description discussed below. Shared data may only be represented by a separate role encapsulating the data and offering services as transactions for data access. There is no way in ROLE to express more fine grained data sharing.

4 Global Interaction Description

Global interaction descriptions are important in the context of a software development methodology, where architectures serve as an intermediate between the requirements and the design. In our framework we start with an interface specification of the future system. In a first step different *contexts*, namely set of roles, of the system behaviour are identified. Then the global interaction *requirements* for each context are specified as *processes*. They determine the interactions between the roles in terms of message flow. At this level, configuration details and interaction control is not tackled. Only the information dependencies of the different roles are made explicit. In a next step the roles are detailed with state space, services, configuration and interaction such that the global interaction requirements are satisfied. Then the actors constituting the system are determined and the set of of roles each actors instantiates. The behaviour of this system is analyzed in order to verify global interaction *properties* as consistency of configuration, deadlock-freedom or performance constraints.

Note that that the transition from processes to roles can be applied at different stages of the software development: during requirements engineering it can be used for enterprise modelling [LK95] and dialog specification, where the former determines the roles of the employees in an organization and the latter determines the role of the software system. The role description of the software system can be used as the starting point for the software design outlined above. In each stage the step can be applied recursively, such that roles are refined into sets of roles.

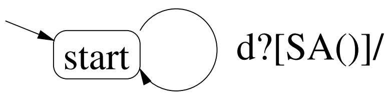
In the following we introduce two different description techniques for global interaction properties and requirements. First we discuss the use of *event traces* for analysis of dynamic properties. Then we show how to use *processes* to define global interaction requirements and how to derive a ROLE architecure satisfying these requirements.

4.1 Global Interaction Properties

There are two kinds of global interaction properties. Static properties can be analyzed without regard to the runtime behaviour of the system. In ROLE configuration consistency can be checked statically: service calls must meet the partner bindings of the service (or role) and bindings of actors must satify the role restrictions. For dynamic properties, it suffices to look at the interaction of *one* role, if this role takes the responsibility for the interaction (like the glue in WRIGHT). In general, however, the global picture must be generated from the local interaction description. For this purpose we employ event traces (also called interaction diagram or message sequence chart) as used in OOSE [Jac92], UML [BRJ96] or Telecommunication applications [IT96, BHS96]. [BHKS97] shows how to use event traces for interaction description between arbitrary components. This technique can easily be adapted to actors used here.

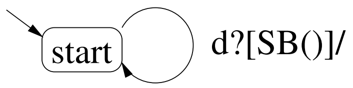
As an example, consider three roles offering one service such that the service of each role must be completed once before one role may accept a new service call. Figure 9 shows the roles of a centralized architecture where the responsibility for the correct sequencing is assigned to a fourth role. In this case the global interaction can be read of the local interaction part of the controlling role.

```
role  A = {
partner d : controller
service SA = {...}
interaction
```



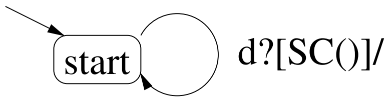

```
}
```

```
role  B = {
partner d : controller
service SB = {...}
interaction
```



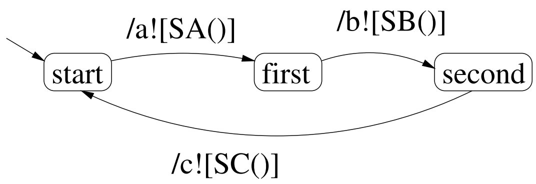

```
}
```

```
role  C = {
partner d : controller
service SC = {...}
interaction
```




```
}
```

```
role  controller = {
partner a : A, b : B, c : C
output request : signal to a ,
    request : signal to b,
    request : signal to c
interaction
```

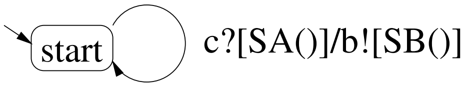



```
}
```

Figure 9: Centralized control

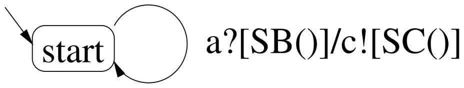
Figure 10 shows the roles of an architecture, where the responsibility is distributed among the components. To verify the required property the set of all possible interaction sequences must be derived from the architecture. Figure 11 shows the corresponding event trace for a system where each role is instantiated with one actor. For each actor a timeline is shown. An arrow starting at a timeline indicates messages sent. An arrow ending at a timeline indicates message acceptance. These diagrams are mostly used to illustrate *typical* interaction sequences. In the context of the tool AUTOFOCUS [HSSS96] we are investigating the possiblities of verifying automata against properties expressed with event traces [EHS97].

role A = {
partner b : B, c : C
output SB() : signal to b
service SA = {...}
interaction



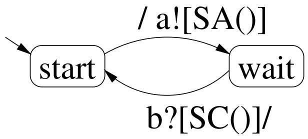
}

role B = {
partner a : A, c : C
output SC() : signal to c
service SB = {...}
interaction



}

role C = {
partner a : A, b : B
output SA() : signal to a
service SC = {...}
interaction



}

Figure 10: Shared control

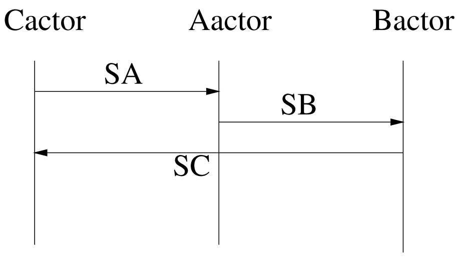


Figure 11: Global Interaction View with event traces

4.2 Global Interaction Requirements

In this section we show how to relate ROLE specifications to global interaction requirements given as *processes*. Processes show the message flow between different roles in reaction to a triggering input. At this level the behaviour of roles is not structured into services, but into *activities*. An activity captures the internal behaviour between an input and the subsequent output.

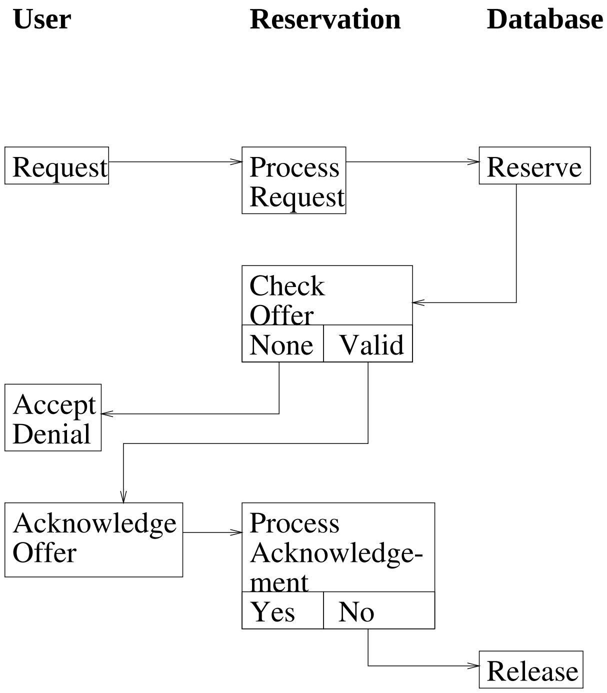


Figure 12: *Reservation* Process

As an example consider figure 12 showing the process of reservation in a car rental company. This process describes the reaction to a customer request for reservation. In this *context* the reservation system consistst of two roles: the reservation_role and the database_role. To accomplish the reservation, the system (in its reservation_role) communicates with the user (who in turn must communicate with the customer. This communication is not shown here.). It tries to reserve a car in the database. If this is not possible, it denies the reservation requests. Otherwise it checks with the user whether the offer is acceptable and accordingly releases the reservation In the picture the involved roles are shown as column headers. Activities are depicted as boxes. Arrows indicate message flow. The process description technique is similar to business process reengineering or workflow descriptions (e.g. [Sch92]). For a more detailed account of the process description technique see [Thu96].
Processes describe global interaction on a high level of abstraction. In particular, they abstract from the realization of the interaction (static vs. dynamic configuration, interaction control) and from the way different processses synchronize on the role data. In

the transition to architecture one changes from this global specification to component-oriented specifications. Taking into account all processes a role is involved in, activities are grouped into services and the interaction part. Activities with write access to the role data must be grouped into services so that data consistency is preserved. As many non-write activities as possible should be grouped together to ensure that the flow of activities determined through the process is reflected as closely as possible in the services. In the example, the reserve and release activity of the database_role must be separate services because of the write access to the data. The reservation_role does not offer services. Instead, it controls the flow of interaction as specified in the process. Thus, all reservation_role activities are grouped together into the interaction part. Figure 13 shows the resulting configuration.

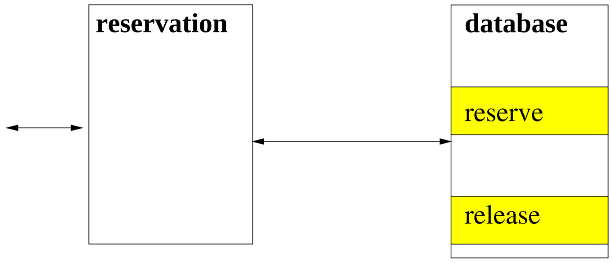


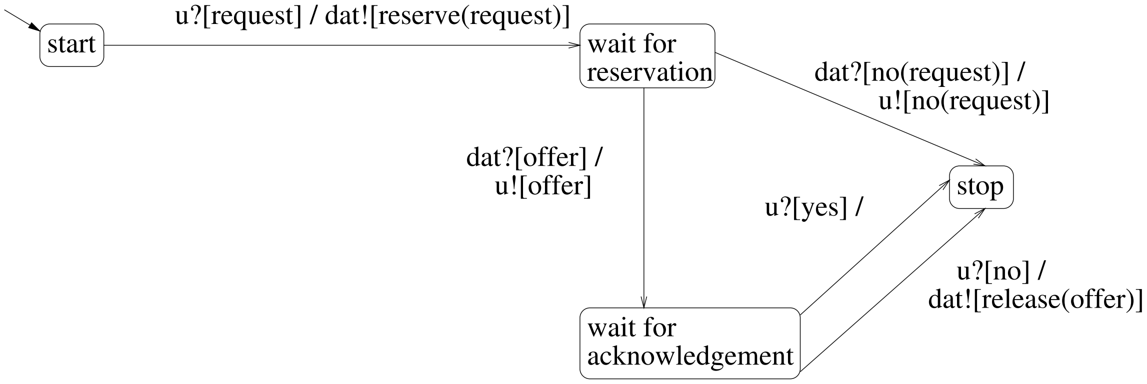
Figure 13: Configuration

Figure 14 shows the corresponding role definitions.
As exemplified above, in our framework global interaction specification is separated from the architecture specification. Processes make explicit the functional requirements on the roles. Architectural decisions like configuration and interaction control are captured with the roles.

5 Conclusions, Related and Future Work

We hope to have shown that roles are a useful paradigm in the realm of interaction description. They make apparent the different contexts of actor interaction. We structure the roles description into state space, services, configuration and interaction to support different levels of abstractions and analysis techniques concentrating on different aspects. We have put the role paradigm into a framework for global interaction description. Event traces are used for analysis of dynamic properties, processes are used for global interaction requirements. We have sketched a methodology relating the different interaction descriptions. Our approach is inspired from work found in the area of object-oriented systems and software architecture. We have discussed the relationship to software architecture research along the way. Related work in object-orientation is discussed below. Also, our approach is not complete. Future enhancements of ROLE descriptions close the paper.

role reservation = {
partner dat : database, u : user
input request : Reservation_Request **from** u, offer : Offer **from** dat,
    in : Bool **from** u
output offer : Offer **to** u, reserve(), release() : signal **to** dat,
interaction



}

role database = {
attributes free : Time $\rightarrow$ Set Car
service reserve = {
    partner res : reservation
    trigger input request : Reservation_Request **from** res
    final output of : Offer **to** res
    post (of = offer(car,time(req)) $\wedge$ car $\in$ free(time(req)) $\wedge$ car $\notin$ free'(time(req))) $\vee$
    (of = no(time(req)) $\wedge$ free(time(req)) = $\emptyset$)}

service release = {
    partner res : reservation
    trigger input offer(car,time) : Offer **from** res
    pre car $\notin$ free(time)
    post car $\in$ free'(time) }
}

Figure 14: Reservation Roles

Related Work

In the realm of object-oriented systems *contracts* were one of the first interaction descriptions proposed. Contracts are similar to *contexts*, in that they specify a set of involved communication participants and their responsibilities within the contract. However, behaviour description of the participants is divided between local descriptions specifying the reaction to service calls and an invariant describing the overall behaviour. Also contracts are used in addition to class specification, such that conformance between classes and participant declaration must be shown. Participants do not have a separate interaction part allowing for synchronization behaviour. Finally, no concurrency is allowed.
Our approach is more closely related to the work of Kristensen [Kri95, KM96] on activities and roles. There, activities are used to make interaction explicit, while roles are only used to partition the state space and services of objects. Thus, roles are component specifications, while activities are connector specifications, which is not distinguished in our framework. Activities may also have synchronization behaviour. However, this behaviour is interleaved with service behaviour, while we only use it to control the service acceptance. The main purpose of activities in Kristensens work is to give an abstract model of the interaction. Therefore also a graphical description of the relationships and control flow of an activity is given. We use processes to give this abstract model, where we additionally abstract from the service structure.
The concept of roles has also been explored in the area of object-oriented database systems [GSR96]. This research is mainly concerned with the classification aspect of roles and does not cover concurrency or synchronization behaviour.
With OORASS [RAB+92] we share the motivation of putting interaction description into a methodological framework. In that work also the problem is separated into several contexts (depicted as role diagrams). The role diagrams only show the reference structure, while we use processes to show the interaction dependencies. The next step in OORASS is to synthesize the different role models for object description using distinguished composition operators. In our framwork roles are independent. For coordination of the different roles of one actor a separate role must be instantiated. To our knowledge no formal definition of OORASS description techniques has been given.

Future Work

We are planning to enhance the flexibility of ROLE in several ways: To describe truely dynamic systems, we want to incorporate dynamic acquisition and abandonment of roles. This, of course, complicates analysis of configuration structure. We also want to incorporate actor classes to allow more succinct system descriptions. We have not yet decided whether to incorporate inheritance between actor classes. It seems that many uses of inheritance can be captured by role instantiation. However, inheritance would be useful on the level of roles (making the concept of role refinement explicit in the ROLE language). Yet another, area of research will be role hierachies where one role is substituted by a whole set of roles.

Besides extension of the description techniques it would be interesting to explore the use of the ROLE framwork for different kinds of interaction description. A prominent example along these lines is the description of design patterns.

Acknowledgement

Thanks are due to Manfred Broy, Christoph Hofmann, Cornel Klein, Ingolf Krüger, Bernhard Rumpe and Monika Schmidt for discussion and helpful comments.

References

[AG94] R. Allen and D. Garlan. Formalizing architectural connection. In *ICSE'94*, pages 71–80. IEEE, 1994.

[BHKS97] Manfred Broy, Christoph Hofmann, Ingolf Krger, and Monika Schmidt. A graphical description technique for communication in software architectures. TUM-I 9705, Technische Universität München, 1997.

[BHS96] M. Broy, H. Hußmann, and B. Schätz. Graphical development of consistent system specifications. In M.-C. Gaudel, editor, *FME'96: Industrial benefit and Advances in Formal Methods*, volume 1051 of *LNCS*, pages 248–267. Springer Verlag, 1996.

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

[Bro95] M. Broy. Mathematical system models as a basis of software engineering. In J. van Leeuwen, editor, *Computer Science Today, LNCS 1000.* Springer Verlag, 1995.

[EHS97] G. Einert, F. Huber, and B. Schätz. Graphical development of distributed systems. submitted to publication, 1997.

[GKRB96] R. Grosu, C. Klein, B. Rumpe, and M. Broy. State transition diagrams. TUM-I 963O, Technische Universität München, 1996.

[GSR96] G. Gottlob, M. Schrefl, and B. Röck. Extending object-oriented systems with roles. *ACM ToIS*, 14(3):268–296, 1996.

[Het96] R. Hettler. Description techniques for data in the SysLab method. TUM-I 9632, Technische Universität München, 1996.

[HHG90] R. Helm, I.M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. In *ECOOP/OOPSLA'90*, pages 169–180, 1990.

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

[HSSS96] F. Huber, B. Schätz, A. Schmidt, and K. Spies. Autofocus - a tool for distributed systems specification. In *FTRTFT'96*, volume 1135 of *LNCS*, pages 467–470. Springer Verlag, 1996.

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

[Jac92] I. Jacobson. *Object-Oriented Software Engineering.* Addison-Wesley, 1992.

[KM96] B.B. Kristensen and D.C.M. May. Activities: Abstractions for collective behaviour. In *ECOOP'96*, volume 1098 of *LNCS*, pages 472–501. Springer Verlag, 1996.

[KRB96] C. Klein, B. Rumpe, and M. Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In E. Naijm and J. Stefani, editors, *FMOODS'96 Formal Methods for Open Object-based Distributed Systems*, pages 323–338. ENST France Telecom, 1996.

[Kri95] B.B. Kristensen. Object-oriented modeling with roles. In *OOIS'95*, pages 57–71. Springer Verlag, 1995.

[Lam94] L. Lamport. The temporal logic of actions. *ToPLAS*, 16(3):972–923, 1994.

[LK95] P. Loucopoulos and V. Karakostas. *System Requirements Engineering.* McGraw-Hill, 1995.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *ESEC'95*, volume 989 of *LNCS*, pages 137–153. Springer Verlag, 1995.

[Nie93] O. Nierstrasz. Composing active objects. In G. Agha, P. Wegener, and A.Yonezawa, editors, *Research directions in concurrent object-oriented programming*, pages 151–171. MIT Press, 1993.

[Pae95] B. Paech. A methodology integrating formal and informal software development. In M.Wirsing, editor, *ICSE-17 Workshop on Formal Methods Application in Software Engineering Practice*, pages 61–68, 1995.

[PR94] B. Paech and B. Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *FME'94, Formal Methods Europe, Symposium '94*, volume 873 of *LNCS*. Springer-Verlag, Berlin, October 1994.

[PW92] D.E. Perry and A.L. Wolf. Foundations for the study of software architecture. *ACM SigSoft, SWE Notes*, 17(4):40–52, 1992.

[RAB+92] T. Reenskaug, E.P. Andersen, A.J. Berre, A. Hurlen, A. Landmark, O.A. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A.L. Skaar, and P. Stenslet. Oorass: seamless support for the creation and maintenance of object-oriented systems. *Journal of object-oriented programming*, (6):27–41, October 1992.

[Sch92] A. Scheer. *Architecture of integrated information systems: foundations of enterprise modelling*. Springer Verlag, 1992.

[SG96] M. Shaw and D. Garlan. *Software Architecture*. Prentice Hall, 1996.

[Sha95] M. Shaw. Architectural issues in software reuse: It's not the functionality, it's the packaging. In *Symposium on Software Reuse*, pages 3–6. ACM Press, 1995.

[Thu96] V. Thurner. A description technique for business process modelling. Internal Report, 1996.

[Yu93] E. Yu. Modeling organizations for information systems requirements engineering. In *IEEE Int. Symp. on Requirements Engineering*, pages 34–41, 1993.