Inf Syst Front (2007) 9:359–373
DOI 10.1007/s10796-007-9040-9

# The adequacy of languages for representing interaction mechanisms

**Remco M. Dijkman · Teduh Dirgahayu · Dick A. C. Quartel**

Published online: 18 July 2007


**Abstract** This paper presents criteria for the adequacy of languages to represent interaction mechanisms. It then uses these criteria to analyse the adequacy of UML. We focus on the interaction mechanisms provided by Web Services technology and by CORBA for request/response, callback, polling and (multicast) message passing. We argue that the criteria for adequacy of a design language are that the language should: (1) be expressive enough to represent the mechanisms; (2) be easy to use when expressing them; (3) be platform independent in the sense that it does not force implementation decisions for a mechanism; and (4) behave corresponding to the mechanisms that it represents. We show that these criteria follow logically from the use of a design language in the design process. For UML we evaluate the first three criteria in a qualitative manner. To evaluate the fourth criteria, we present Coloured Petri Nets that capture the behaviour of both the mechanisms precisely and the UML constructs that represent them. Subsequently, we check the correspondence of their behaviour.

This work is part of the Freeband A-MUSE project. Freeband (http://www.freeband.nl) is sponsored by the Dutch government under contract BSIK 03025.

R. M. Dijkman (✉)
Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: r.m.dijkman@tm.tue.nl

T. Dirgahayu · D. A. C. Quartel
University of Twente,
P.O. Box 217, 7500 AE Enschede, The Netherlands

T. Dirgahayu
e-mail: t.dirgahayu@utwente.nl

D. A. C. Quartel
e-mail: d.a.c.quartel@utwente.nl

**Keywords** Design language · Design concept · Communication patterns · Middleware

## 1 Introduction

Middlewares are defined to make the lives of developers easier, by providing re-usable implementations of advanced interaction mechanisms. Examples of such mechanisms are: remote procedure calls, transactions, publish/subscribe mechanisms, negotiations and long-running business-to-business interactions. Similarly, design languages could make the lives of designers easier, by providing re-usable design concepts that represent these advanced interaction mechanisms. Such design concepts help to:

- simplify designs, by providing a single concept, or a small collection of concepts to represent an advanced interaction mechanism;
- transform designs to implementations, by providing abstract (platform independent) concepts of which the transformation to (various) middlewares is clear; and
- analyse the correctness of the design in early stages of the design process.

To be able to analyse the correctness of a design the design concepts must properly reflect the relevant properties of the represented interaction mechanisms. In contrast, if concepts do not reflect the properties of their middleware counterparts faithfully, this may lead to wrong conclusions during analysis. For example, the concept of reliable message passing does not behave like message passing middleware, in which message passing is typically unreliable. This may cause designers to conclude wrongly that a message will always be received by the receiving side and therewith that the implemented business transaction will be

successfully completed. This leads to problems if the concept is used in designs in which reliability is an issue (such as the design of a banking system).

The first goal of this paper is to present and motivate the criteria for adequacy of advanced interaction design concepts (and design concepts in general) and to show how these criteria can be checked. The second goal of the paper is to evaluate the adequacy of UML for representing advanced interaction mechanisms, based on the criteria.

To motivate the criteria for adequacy of design concepts, we analyse the role that design concepts play in the design process.

To analyse the adequacy of UML for representing advanced interaction mechanisms, we analyse interaction mechanisms implemented in existing middleware and the concepts that UML 2.0 (OMG 2003; OMG 2004) provides to represent interaction mechanisms. To capture their properties precisely, we define both the interaction mechanism and the concepts using Coloured Petri Nets. Subsequently, we evaluate the adequacy of the design concepts from UML 2.0 for representing the analysed interaction mechanisms, based on the criteria that are the result of the first step.

We focus on the mechanisms that CORBA (OMG 2002b) and Web Services (W3C 2004) provide for request/response, callback, polling and (multicast) message passing.

This paper is further structured as follows. Section 2 explains the basic properties of a design process and motivates the criteria that design concepts in the design process should meet. Section 3 analyses interaction mechanisms in CORBA, and models them in terms of Coloured Petri Nets. We refer to Dijkman et al. (2006a) for a similar analysis of Web Services interaction mechanisms. Section 4 analyses the UML 2.0 model elements to represent interactions and models them in Coloured Petri Nets. Section 5 evaluates if and how the interaction mechanisms from Sections 3 and 4 can be represented using UML 2.0. Section 6 presents related work and Section 7 discusses our conclusions and future work.

## 2 Criteria for design concepts

We derive the criteria for design concepts from the role that design concepts play in a design process. Figure 1 illustrates a typical design process. During a design process we gradually transform a set of requirements into a design that is detailed enough to be implemented. The process starts out with an initial design, which is a rough sketch of the properties of the implementation, taking only the most important requirements into account. We refine this design by taking more requirements into account. Based on these requirements we decide on other properties of the implementation. However, properties that we already decided on should not be violated. We say that a design is at a lower level of abstraction than another design if it considers more implementation properties. We say that it is at a higher level of abstraction if it considers less implementation properties.

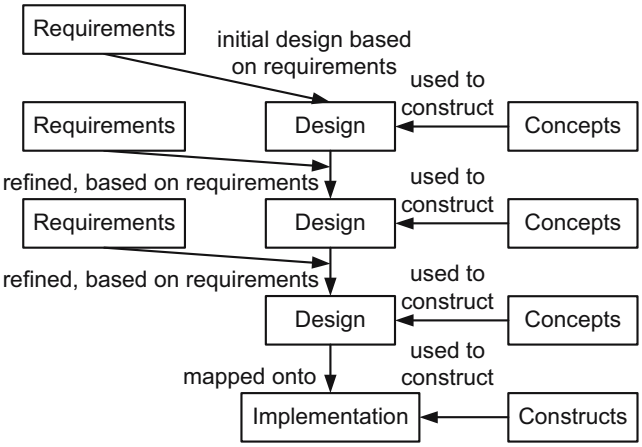



**Fig. 1** Interaction concepts and design

We can construct a design at each level of abstraction by composing instances of available design concepts. A design concept is an abstraction of some common and essential properties of the implementation. Hence, design concepts determine which implementation properties we can represent and how well we can represent them. Therefore, design concepts must:

1. be *expressive* enough to represent the properties that we consider relevant at the level of abstraction at which we want to use them;
2. represent these properties in a *suitable* manner (i.e.: allow a designer to represent properties in a manner that is easy to use and understand);
3. be *platform independent* in the sense that they do not favour some design decisions over others;
4. be *faithful* to the implementations that they are meant to represent (i.e. do not cause a designer to represent an implementation in a way that the implementation does not behave).

We discuss each of these criteria below in more detail. We also provide operationalizations of the criteria: 'rules of thumb' to check if a set of concepts meets the criteria.

### 2.1 Expressiveness of design concepts

To decide if a set of concepts is expressive enough, we must answer the question which properties are 'relevant' at a certain level of abstraction. For a part this is up to the designer to decide. However, 'best design practices' exist that suggest that certain properties are used at a certain level

of abstraction. Hence, we can look for these best practices to discover relevant properties. We consider two forms of recording best design practices: design patterns and middleware implementation constructs.

In the terminology that we use here, a *design pattern* is a well-known set of properties that satisfies a given requirement. The ability of a set of design concepts to represent such patterns, or compositions of properties, is often used as a criterion for the expressiveness of a language.

A design process is executed with the goal of eventually implementing the design. Therefore, at least in theory, a level of abstraction exists at which the design can be mapped onto an implementation. Design concepts at this level of abstraction can be mapped automatically onto implementation constructs. Such a level of abstraction may only exist in theory, because a design may never actually be constructed at this level. The design process may end at a level of abstraction at which refinement is still needed to construct an implementation. However, even if this is the case, there always is a (refinement) relation between the concepts at the lowest level of abstraction that is actually used and the implementation constructs.

*Operationalization* Check if the concepts can represent the most common interaction patterns: 'synchronous request/ response', 'asynchronous request/ response: callback', 'asynchronous request/ response: polling', 'one-to-one message passing' and 'multicast message passing'. Often the names of the concepts will reveal if the interaction mechanism is considered. For example, if a concept with the name request/response exists, the (synchronous) request/ response mechanism can be represented. Note that at this stage we are only concerned with the question whether a concept *can* be represented, not whether a concept can be represented properly. The other criteria will deal with that.

Optionally, support for other interaction patterns from literature (Barros et al. 2005; Hohpe and Woolf 2004; Ruh et al. 2001) or from a particular middleware can be checked. Also, support requirements can be adapted for a particular organization or design project.

## 2.2 Suitability of design concepts

A set of concepts is suitable for representing some property if that property can be represented using one concept or a small composition of concepts. In contrast, we say that a set of concepts is *unsuitable* for representing a property if a composition of a large number of concepts is needed to represent that property. Such a set of concepts is unsuitable, because it leads designs that are unnecessarily complex, even to the extent that they become impossible to understand.

As an example consider a set of design concepts that only supports message passing as an interaction concept. In this case, if a designer wants to represent an advanced interaction, such as a polling interaction, the designer has to represent that interaction using a composition of message passing concept instances. Moreover, the designer has to use this representation for each polling interaction. This yields unnecessarily large designs that become hard to understand because of their size.

To have a more suitable set of design concepts, a design language can allow for the definition of composite concepts. A composite *concept* is a concept that consists of other concepts (Quartel et al. 2005). Using composite concepts, we can introduce a single concept to represent a composition of a large number of other concepts. Hence, making the set of concepts more suitable.

*Operationalization* Check if each relevant interaction pattern (identified under the first criterion) can be represented by a single concept or a suitable composition of concepts. As a rule of thumb a suitable composition consists of up to four concept instances. However, in case we can use a 'composite concept', the number of concepts can be arbitrary.

## 2.3 Platform independence of design concepts

A set of design concepts is platform independent (at a certain level of platform independence) if it does not force the designer to make design decisions that he does not (yet) want to make. A set of design concepts can force a designer to make certain design decisions, by:

1. limiting design choice;
2. encouraging a certain design decision, while discouraging another;
3. forcing the designer to make a design decisions in combination.

We say that a set of concepts *limits the design choice* of a designer, if there is no concept that represents some property of a certain pattern or implementation construct. Such a set of concepts limits design choice, because it is likely that the designer will not use the property. An example of a limitation of the design choice of a designer exists in the UML component model, because the UML component model does not allow ports of a component to be created independently of the creation of the component itself. This at least discourages a designer from creating a design in which ports of a component are created independently of that component (say for each connection that a client establishes with that component). It also

discourages an implementer from using middleware constructs that allow for the creation of ports independently of a component. Such constructs are, for example, allowed in the CORBA Component Model (CCM; OMG 2002a).

We say that a set of concepts *encourages a designer to make certain design choices* if representing some design patterns or implementation constructs is more easy than representing others. We claim that design concepts should be neutral with respect to what the designer wants to represent. Of course this is only possible to a certain extent; design patterns and implementation constructs that are more involved are typically harder to represent as well. This is a problem of the design patterns and implementation constructs themselves, not a problem of the design concepts that represent them.

We say that a set of design concepts forces a designer to *make certain design choices in combination* if the designer, by using some concepts to represent some properties, necessarily has to specify other properties as well, while he did not want to commit to these properties yet.

We argued in previous papers that different levels of platform independence can be distinguished, depending on the choices that have been made with respect to the target platform (Andrade Almeida et al. 2006). For example, at some level of platform independence, the designer may want to specify that an interaction is implemented using a publish/subscribe mechanism, but not yet the specific middleware that will be used. However, at a higher level of abstraction, the designer may want to specify that a multicast message passing interaction is required, but not yet the mechanism that will be used to provide that interaction (e.g. publish/subscribe can be used, but also a multicast mechanism in which the message sender specifies the set of desired recipients).

*Operationalization* This criterion should be checked for each concept, by careful comparison of the concepts and the properties (of the interaction mechanism) that it represents. A rule of thumb cannot be given here.

### 2.4 Faithfulness of design concepts

We say that a design concept is faithful, if it behaves like the implementation construct onto which it is mapped or the design pattern that it represents. In case a design concept is related to an implementation construct or design pattern by refinement, it must behave as an abstraction (the inverse of refinement) of that construct or pattern.

Although a designer may not always have a mapping in mind, design concepts can be suggestive with respect to their potential mapping. For example, a design concept representing a remote operation call suggests that it is mapped onto an implementation construct that has a similar behavioural pattern (i.e. a request is sent in one direction after which a response can be sent in the other direction). Moreover, *some* mapping must be possible for a design concept, but we will show below that this is not always the case.

We say that a concept is *unfaithful to a mapping* if it does not behave according to the implementation construct or design pattern to which it is related. A problem associated with this mismatch is that, if a concept is unfaithful to its mapping, analysis of a design that uses the concept leads to wrong conclusions about the implementation. For example, if a message-passing concept assumes that communication is reliable, while the implementation construct onto which it is mapped implements unreliable communication, then the designer can draw incorrect conclusions of whether a message arrives at the receiver or not. If a concept is unfaithful to a mapping, the problem can be solved either by changing the behaviour of the concept or by changing the mapping.

We say that a concept is *unfaithful to any mapping* if it is impossible to construct a mapping onto an implementation construct that behaves according to the concept. This problem exists for concepts that represent reliable message passing, because it has been shown that reliable message passing cannot be implemented (Lamport et al. 1982).

In this paper we check faithfulness of UML 2.0 concepts to middleware implementation constructs in CORBA and Web Services, by constructing formal models of implementation constructs and the UML using Coloured Petri Nets. Subsequently, we employ the formal notion of refinement that we developed in previous work (Dijkman et al. 2006b; Quartel et al. 2002) to check the relation between the concepts and the constructs.

*Operationalization* Check faithfulness of each design concept to the interaction mechanism that it represents, by checking whether the interaction mechanism is a correct refinement of the design concept (proving or disproving that the concept is a correct abstraction of the mechanism). To be able to do this you need a formalization and a notion of refinement.

## 3 CORBA interaction mechanisms

CORBA specifies re-usable mechanisms for interaction between software entities. We focus on the mechanisms that it provides for:

- synchronous request/response communication;
- asynchronous request/response communication, based on callback;
- asynchronous request/response communication, based on polling;

– one-to-one message passing (also called one-way operations in CORBA);
– multicast message passing, based on publish/subscribe (also called the Notification Service in CORBA).

We represent the externally observable behaviour of each of these mechanisms, as it is observed by the entities that use them to interact, abstracting from how the middleware implements these mechanisms.

### 3.1 Synchronous request/response

In request/response communication, we distinguish a *client*, which sends a request and receives a response, and a *server*, which receives a request and sends a response. A response can either be a normal response or an exception that notifies the client that some exceptional situation has occurred. Exceptions can either be returned by the server (in which case we call it a *user exception*), notifying the client that the server could not process the request, or by the middleware (in which case we call it a *system exception*), notifying the client that a problem occurred in the communication. In case a system exception occurs, the middleware notifies the client whether the server completed processing (got to the point where it returns a response). For that purpose the middleware returns a 'yes', 'no' or a 'maybe'. The 'maybe' represents that it cannot be sure.

Figure 2 presents a Coloured Petri Net that represents the observable behaviour of a synchronous request/response mechanism. 'C_KIND' represents the kind of exception

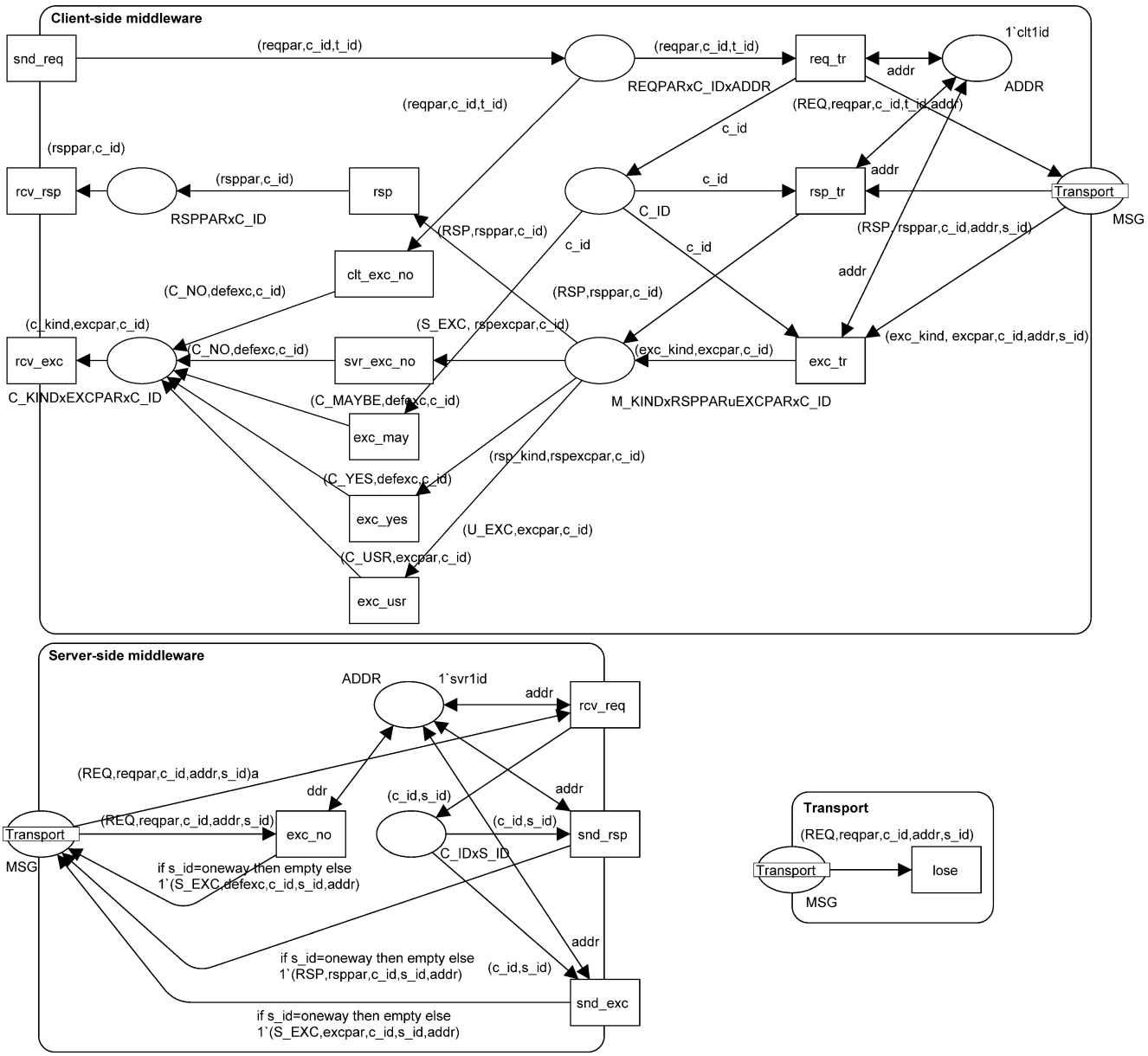



**Fig. 2** CORBA request/response

returned. It can take the values: 'C_NO' representing that the server did not complete processing; 'C_YES' representing that the server did complete processing; 'C_MAYBE' representing that the client cannot be sure if the server completed processing; or 'C_USR' representing that the server returned a user exception.

At the client-side, a client sends a request, which is associated with an id ('c_id') and an address at which the server can be located ('t_id'). The client's request is either sent ('req_tr') to the server as a request ('REQ') message or an exception is returned to the client ('clt_exc_no'), notifying the client that the request could not be sent. The request message is annotated with the client's address, to which the response can be directed by the server. Once the request is sent, the client-side waits for a response or exception message from the server. The reception of a response or exception results in an internal event that represents: the reception of a response ('rsp'); the reception of a server-side system exception ('svr_exc_no'); the occurrence of a client-side system exception upon receiving the response or exception ('exc_yes'); or the reception of a server-side user exception ('exc_usr'). If a response or exception was not received within some timeout, the client-side middleware generates an exception ('exc_may'). This exception disables the reception of a response or exception for the request/response invocation.

At the server-side, a request message is received by the server. The address to which the message is targeted must match the server's address. If the message could not be processed an exception is generated ('exc_no') and returned to the client. Otherwise, the request is delivered to the server, which either responds with a response or with an user exception.

(Multiple) clients and servers can be connected by connecting the 'client-side' and 'server-side' parts from Fig. 2 to the same 'transport'. The transport is unreliable and may deliver messages in a different order than the order in which they were sent (the CORBA specification explicitly states that "end-to-end ordering guarantees cannot be made" (OMG 2002b; page 22–55).

### 3.2 Asynchronous request/response: Callback

In CORBA, for asynchronous Request/Response mechanisms, nothing changes at the server-side. At the client-side, using the callback mechanism, the client sends a request with which it passes an address at which it expects the response. At this address a reply-handler must be active that receives the response. After the client has sent the request, it can continue processing. Sending the request may cause a system exception at the client-side. CORBA does not

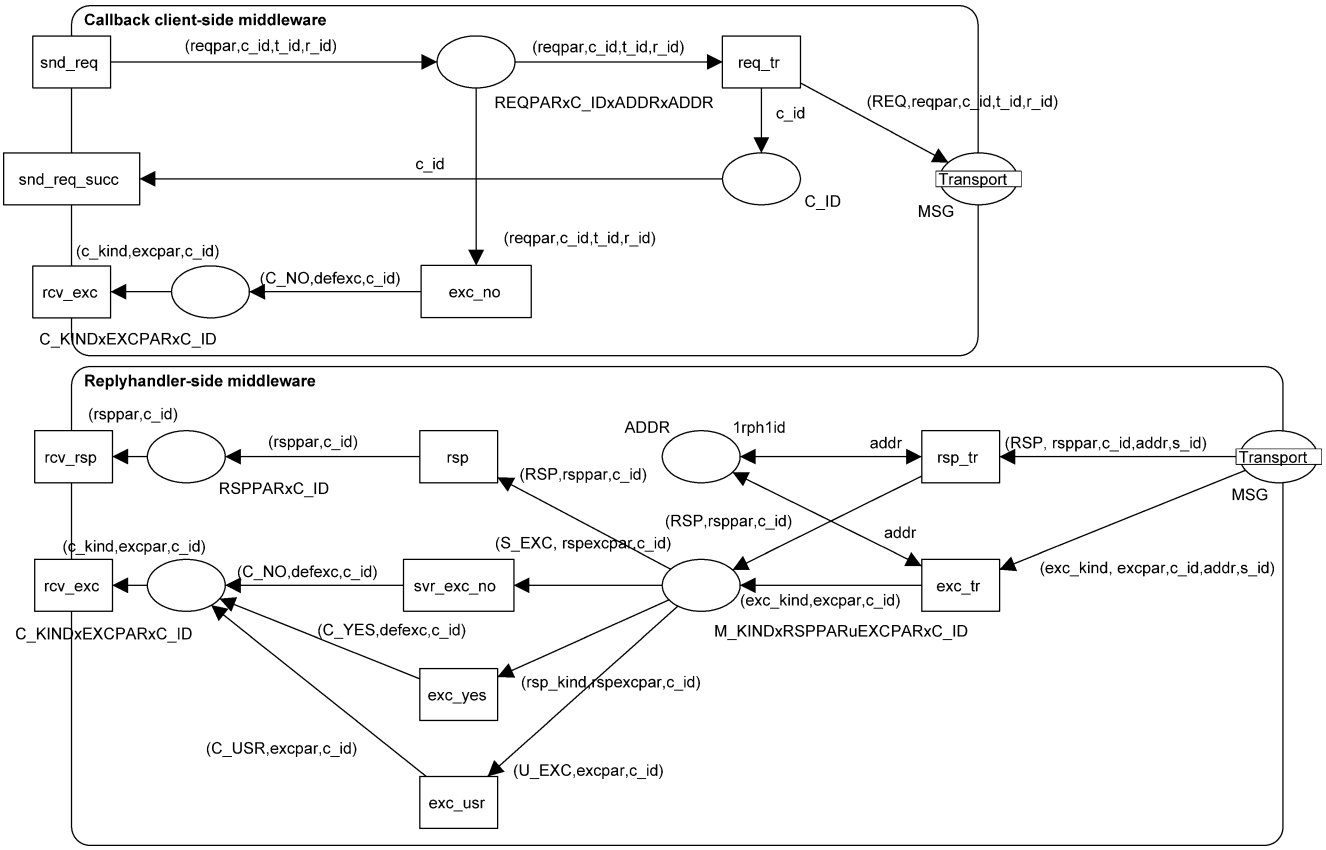



**Fig. 3** CORBA callback

automatically provide the threading mechanisms that ensure that the reply-handler is ready to handle a response at the same time that a client is handling a request. Therefore, the implementer must implement appropriate threading strategies. In terms of the model: the client-side and the reply-handler side from Fig. 3 *can* operate independent of each other, but the implementer must make sure that they *do*.

Figure 3 presents a Coloured Petri Net that represents the observable behaviour at the client-side and the reply-handler side. The Petri Net is similar to the Petri Net from Fig. 2, but split-up into a client and a reply-handler part. When sending a request, the client side passes the address of the reply-handler ('r_id') as the address where the response should be directed. The client-side middleware then notifies the client whether the request was sent ('snd_req_succ') or not ('rcv_exc'). 'maybe' exceptions are not handled for callback invocations.

(Multiple) clients, reply-handlers and servers can be connected by connecting the 'client-side' and 'reply-handler' parts from Fig. 3 and the 'server-side' part from Fig. 2 to the same 'transport'.

### 3.3 Asynchronous request/response: Polling

At the client-side, using the polling mechanism, the client sends a request, upon which it receives an instance of an abstract data type that it can poll for the response. Figure 4 presents a Coloured Petri Net that represents the observable behaviour at the client-side. The figure shows that after the client sends a request, the client-side middleware notifies the client whether the request was sent ('snd_req_succ') or not ('rcv_exc_req'). With a success notification, the client is returned an identifier ('c_id') for the invocation with which it can poll ('poll') for the response. If the client polls for a response, it can receive: a notification ('poll_timeout') that the response could not be obtained within some timeout period; a notification ('poll_noobject') that the response was already obtained by the client; a notification

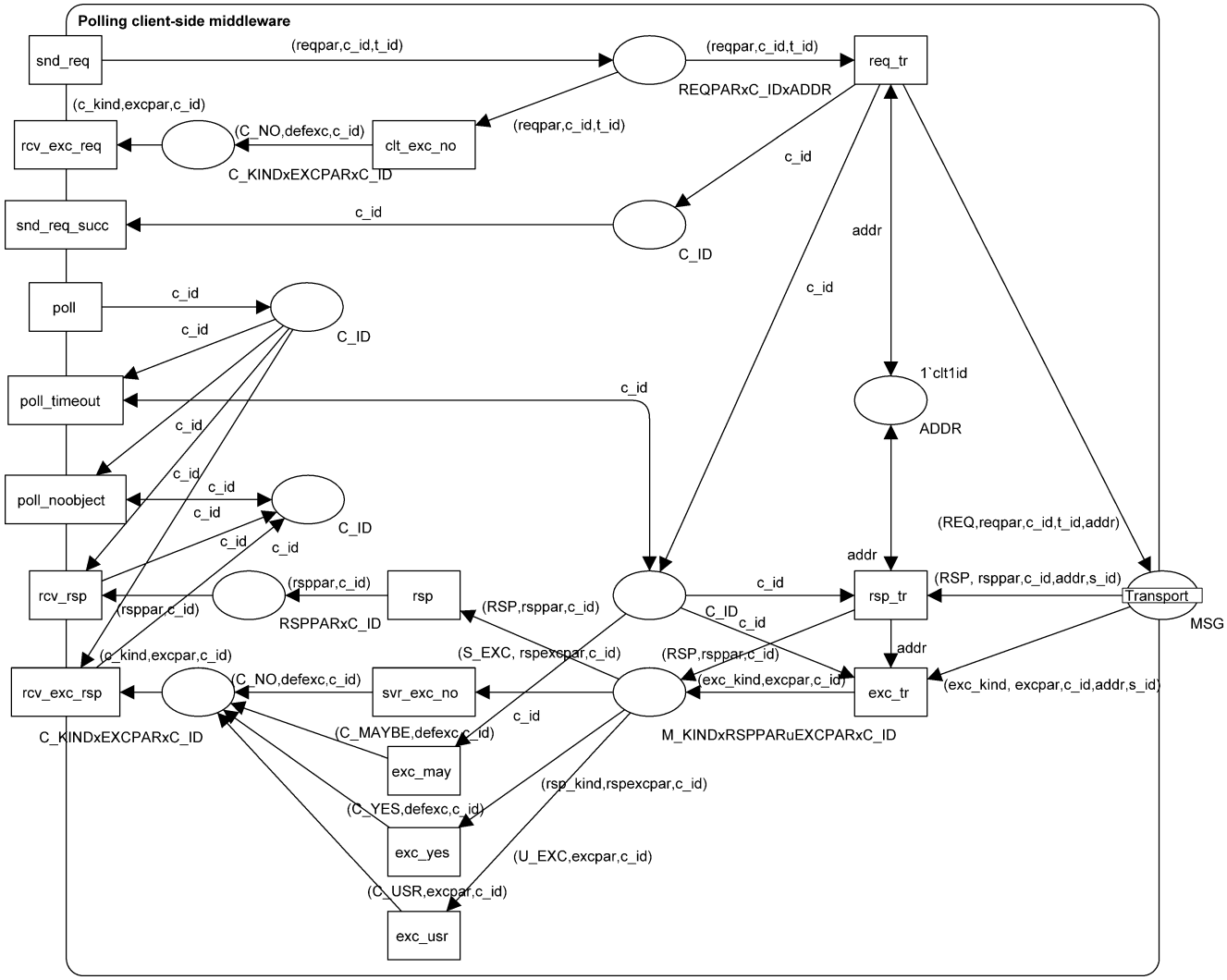



**Fig. 4** CORBA polling

('rcv_rsp') carrying the response; or a notification ('rcv_exc_rsp') carrying an exception.

### 3.4 One-to-one message passing

In one-to-one message passing mechanisms, one party can send messages to another.

In CORBA, this mechanism is implemented using 'one-way' request/response communication. For this mechanism, nothing changes in the server represented in Fig. 2. However, the server will not send a response upon receiving a request for 'one-way' communication. At the client side, mechanisms for dealing with responses and exceptions that occur after the request was sent, can be removed.

### 3.5 Multicast message passing

CORBA implements multicast message passing using a publish/subscribe mechanism. Publishers can either 'push' messages to a broker themselves, or the broker can continuously 'pull' messages from the publisher. Similarly, subscribers can 'pull' messages from the broker themselves, or have the broker 'push' messages to them as soon as they are available.

Figure 5 represents the behaviour of the publishers and subscribers in CORBA. A publisher can either 'push' an event to a channel. After this the publisher can either receive an exception ('rcv_exc'), representing that there was an exception sending the event to the channel, or a response ('rcv_rsp'), representing that the event was successfully delivered to the channel. A 'pull' can also be initiated by the channel. After this the publisher can respond by sending the event to the channel ('snd_rsp'), or by indicating that an event cannot be sent to the channel ('snd_exc'). Even if an event is sent by the publisher, the event may be lost on its way to the channel.

Similarly, the channel can initiate a 'push' on the subscriber. After this the subscriber can respond by indicating that he received the event ('snd_rsp'), or by indicating that he does not want to receive the event ('snd_exc'). Also, the client can try to 'pull' an event from the channel. This can result in a notification from the channel that no events could be pulled ('rcv_exc'), or a notification with the pulled event ('rcv_rsp').

## 4 UML interaction concepts

This section represents the behaviour of the UML concepts that we can use to represent: request/response and one-to-

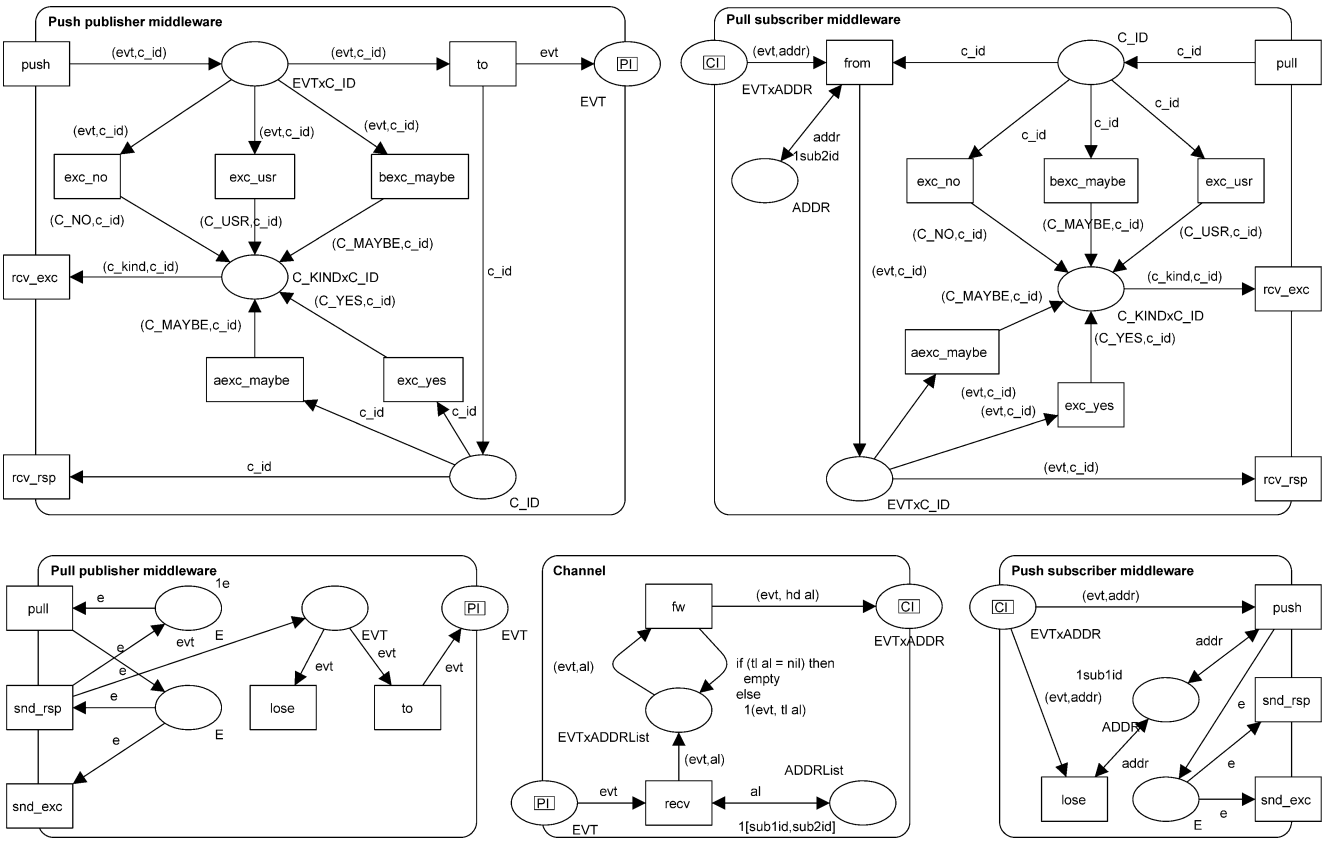



**Fig. 5** CORBA multicast message passing

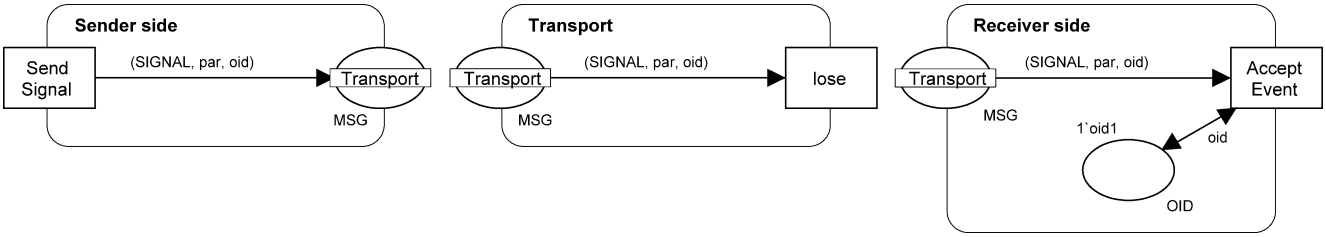



**Fig. 6** Sending and receiving a signal

one message passing. UML also has a mechanism for broadcast message passing. However, we do not consider this mechanism, because broadcast is not addressed by the middlewares that we investigated (only multicast is).

The UML 2.0 provides two different ways to represent request/response and one-to-one message passing: by signal passing and by operation call. An operation call can be either asynchronous or synchronous.

The UML defines the properties of the communication medium that transports the requests, responses and messages as a semantic variation point. These properties include transmission delays, loss of requests, reordering and duplication. In this paper we choose an unreliable medium that does not preserve ordering, because those properties are common properties for communication mechanisms underlying the request/response and message passing mechanisms.

### 4.1 Signal passing

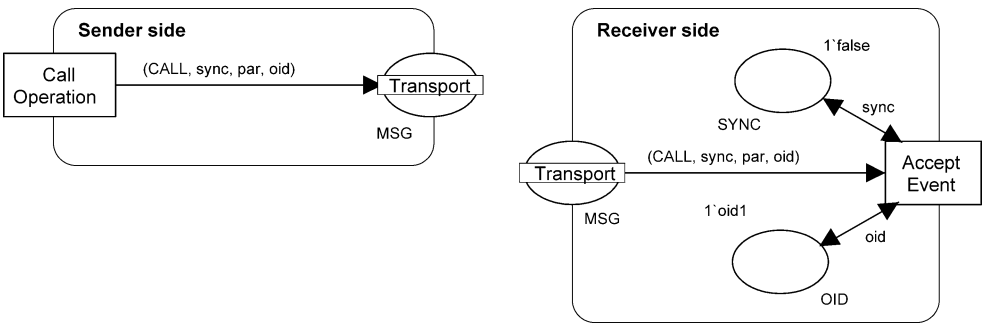
Sending a signal can only be done asynchronously. Figure 6 presents a Coloured Petri Net that represents the behaviour of sending and receiving a signal. A sender sends a request message as a signal to a receiver by executing a SendSignalAction. The 'SendSignal' transition represents this action. This action creates a 'SIGNAL' message that contains input parameters supplied by the sender 'par' and the receiver's identity 'oid'; and sends the signal to the receiver through some communication or transport medium. The action then completes immediately. The transport medium transports the request from the sender to receiver, but may lose the message. To receive the signal, the receiver executes an AcceptEventSignal. The 'AcceptEvent' transition represents this action. This action waits for the occurrence of an event. In this case, the action waits for a signal reception event that meets the receiver's identity. The receiver's identity is stored on the place with the initial marking 'oid1'. When a matched event is detected, the action completes and produces outputs describing the occurrence of the event and the values carried by the signal 'par'.

### 4.2 Asynchronous operation call

Figure 7 presents a Coloured Petri Net that represents the behaviour of an asynchronous remote operation call. To make an asynchronous operation call, a sender executes a CallOperationAction with attribute isSynchronous sets false. The 'CallOperation' transition represents this action. This action creates a 'CALL' message that contains an indication whether the call is synchronous 'sync', the input parameters supplied by the sender 'par' and the receiver's identity 'oid'. The action then sends the message to the receiver through some transport medium and completes immediately.

To accept an asynchronous operation call, the receiver executes an AcceptEventAction. The 'AcceptEvent' transition represents this action. This action waits for the occurrence of an event. In this case, the action waits for an asynchronous call reception event that meets the receiver's identity. The identity is stored on the place with the initial marking 'oid1'. This action cannot be used to receive a synchronous operation call. When a matched event is detected, the action completes and produces outputs describing the occurrence of the event and the values carried by the message 'par'.

**Fig. 7** Asynchronous remote operation call



### 4.3 Synchronous operation call

Figure 8 presents a Coloured Petri Net that represents the behaviour of a synchronous remote operation call. To make a synchronous operation call, a sender executes a CallOperationAction with attribute isSynchronous sets true to make the action waits for a reply or an exception after sending a request. The 'CallOperation_snd', 'CallOperation_rcv' and 'CallOperation_exc' transitions represent this action. The 'CallOperation_snd' transition handles the sending of a request message, while the 'CallOperation_rcv' and 'CallOperation_exc' handle the reception of a reply and an exception message respectively.

This CallOperationAction creates a 'CALL' message that contains an information 'REQ' to indicate that the message is a request, the identity of the call 'cid', an indication whether the call is synchronous 'sync', the input parameters supplied by the sender 'par', the receiver's identity 'roid' and the sender's identity 'soid'. The action sends the request to the receiver through some transport medium and then waits for a reply or an exception from the receiver. To correlate a reply or an exception message with a request message, the identity of the call 'cid' is stored.

To accept a synchronous operation call, the receiver executes an AcceptCallAction. The 'AcceptCall' transition represents this action. This action waits for the occurrence of a synchronous call reception event that meets the receiver's identity. The identity is stored on the place with the initial marking 'oid1'. When a matched event is detected, the action completes and produces outputs describing the occurrence of the event, the values carried by the message 'par' and information necessary for returning a reply to the sender.

This action can also be used to receive an asynchronous operation call, but no corresponding reply or exception can be sent for this operation call.

To send a reply, the receiver executes a ReplyAction. The 'Reply' transition represents this action. This action receives input parameters 'par' and, by using information provided from its corresponding request, this action creates a reply message to be sent to the sender. An information 'RSP' is used to indicate that the call message is a reply.

If the execution of the called operation raises an exception, an exception message is transmitted back to the caller. The 'send exception' transition represents the sending of the exception message to the caller. It should be noted that the UML 2.0 does not specify how exception messages are transmitted to the caller. Thus the transition does not correspond to any action in the UML 2.0.

After sending a request message, the CallOperationAction waits for a reply reception event that meets the request sender's identity and a call identity 'cid'. The sender's identity is stored on the place with the initial marking 'oid3'. When a matched event is detected, the action completes and produces outputs describing the occurrence of the event and the values carried by the message 'par'. When an exception message is received, it raises an exception in the execution of the CallOperationAction.

## 5 Representing interactions in UML

In this section we discuss the adequacy of UML for representing the CORBA interaction mechanisms analysed above and the Web Service interaction mechanisms explained by Dijkman et al. (2006a). To analyze the faithfulness of the UML model elements with respect to the

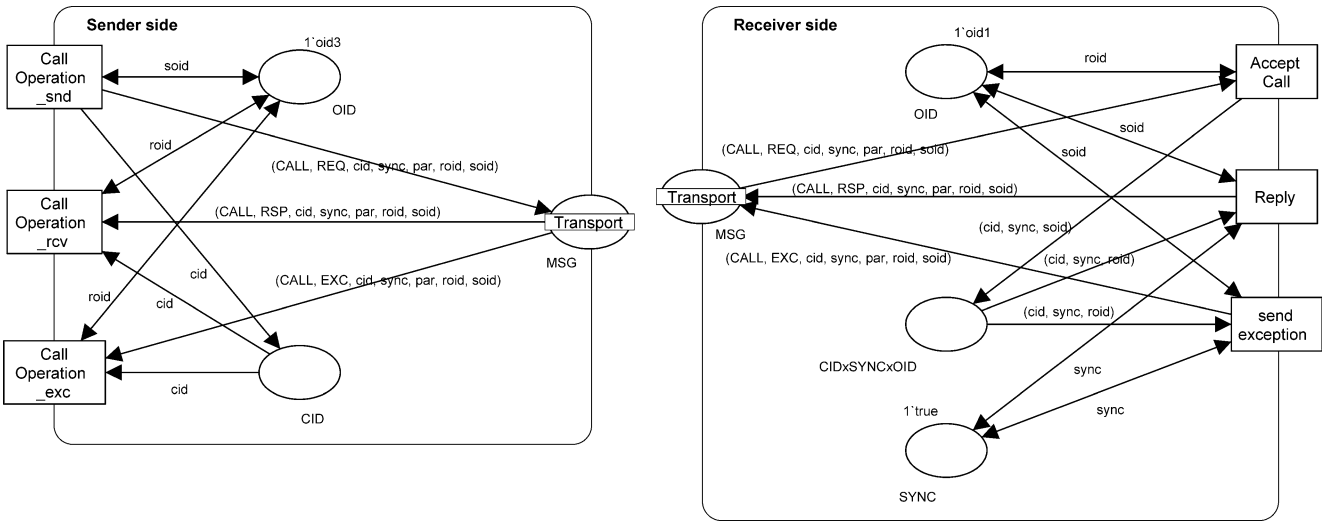



**Fig. 8** Calling a synchronous operation call and receiving a reply or an exception

interaction mechanisms, we check if the UML model elements are a correct abstraction of the model elements. To perform this check we use the Petri nets from the previous sections and from Dijkman et al. (2006a) and the refinement theory explained by Dijkman et al. (2006b) and Quartel et al. (2002). Although Dijkman et al. (2006a) provide algorithms for checking refinement directly on Petri nets, we explain the refinement checks using the execution traces of the Petri nets, because explaining the algorithms for checking refinement is out of the scope of this paper.

### 5.1 Representing request/response

We represent request/response interactions, using the UML synchronous operation call that communicates through an unreliable medium. The models for that interaction are presented in Section 4.3. We claim that this is the suggested mapping from the UML synchronous operation call onto the CORBA and Web Services request/response constructs. What else should the UML synchronous operation call with unreliable transport be mapped onto?

A UML operation does not support the notification of communication failure. Hence, it is *unfaithful* to an interaction mechanism that notifies such failures, such as a mechanism that throws exceptions when a request cannot be sent or when a request or response is lost. In the Petri net execution traces this becomes apparent, because the CORBA request/response allows for the trace: 'snd_req', 'clt_exc_no', 'rcv_exc', in which a request is directly followed by an exception, because the request could not be sent via the communication medium. This trace from the CORBA model corresponds to the following trace from the UML model: 'Call Operation_snd', 'Call Operation_rcv'. However, UML does not allow this trace nor other traces in which system exceptions are returned. Similar conflicts exist for cases in which a response cannot be received and for cases in which a request or response cannot be (de-) marshalled.

We can solve the unfaithfulness problem by modelling a timeout mechanism that detects message loss after a certain timeout (Quartel et al. 2005). However, explicitly modelling this mechanism limits the *suitability*, because it requires that the mechanism is modelled for each interaction, leading to extensive models. It also limits the *platform independence* of UML, because it reveals implementation details about how communication failures are detected, namely by a time-out mechanism, while communication failures may be detected in other ways.

### 5.2 Representing callback

To represent callback we use a composition of asynchronous UML operations, one that represents the request and one for each possible response and exception. The request has to carry the address of the object that will handle the response. The modeller has to ensure that requests and responses can be dealt with simultaneously in the UML model, in this way modelling a threading strategy, like the implementer has to ensure this threading strategy for the CORBA implementation. We consider this way of representing the callback mechanism *suitable*, because only a small collection of UML concepts is needed to represent the callback mechanism.

This way of representing can limit *faithfulness*, because it allows the trace: 'Call Operation (..., server)', 'Accept Event', 'Call Operation(..., client)', 'Accept Event', 'Call Operation(..., client)', 'Accept Event', .... This trace could be useful, for example, for a client to announce its address to the server and for the server to subsequently stream a video to the client. However, the trace cannot be performed by the CORBA nor by the Web Services callback mechanism. Hence, to allow for a faithful mapping, the behaviour of the server in the UML model must be restricted, such that it does not perform this trace.

Also, we claim that this representation limits *platform independence*, because:

1. user exceptions and responses cannot be declared as such, but have to be specified as asynchronous operation calls, meaning that in UML we encourage an implementation onto multiple asynchronous operation calls and discourage an implementation onto a middleware's callback mechanism; and
2. it requires the server-side to be aware that it is being called using a callback operation, while in the CORBA implementation this is not necessary.

To solve problem 1 we can stereotype the operations. However, then we do not have a 'pure' UML model anymore.

### 5.3 Representing polling

We represent the remote polling mechanism in UML by a UML synchronous operation call. The operation call must have an additional parameter to specify the 'id' of the response for which we are polling. We cannot distinguish this representation of remote polling from a regular remote operation call, unless we use a stereotype. Hence, it limits *platform independence*.

To represent local polling we need to model an intermediary object that:

1. accepts a call from the client to send the request;
2. sends the desired request to the server;
3. returns control to the client;
4. awaits the response from the server;

5. accepts a poll from the client to return the response;
6. upon a poll, sends the response if it is available.

We claim that, similar to the callback mechanism, this representation limits *platform independence*. However, the problem with the polling mechanism is more serious, because the designer has to make choices regarding the implementation of the intermediary object. These choices do not necessarily reflect the choices that are made by Web Services or CORBA. Hence, there may be a mismatch between the implementation and the design. The choices that we made to represent the polling mechanism are that:

1. the intermediary object exists locally at the client side and invokes the server that is remote;
2. the intermediary object sends a request to and receives a response from the server by performing synchronous call; and
3. the intermediary object implements some threading mechanism to allow the client to continue processing, while it processes the synchronous request/response to the server.

Another drawback of this solution is that the call from the client does not address the server, but the intermediary object. We can construct a solution in which the client directly addresses the server and the intermediary object only handles the response. However, this solution has the drawback that the client must obtain both the address of the intermediary object and the address of the server object. Also it has the drawback that the server has to change, because it has to obtain requests from the client and send responses to the intermediary object.

Moreover, modelling a complete mechanism causes the model to expand, leading to *unsuitability* of the resulting model. The benefit of modelling a complete mechanism is that (provided the modelling language is expressive enough) it can be modelled to be completely *faithful* to the mechanisms that it represents.

### 5.4 Representing one-to-one message passing

We can represent one-to-one message passing in UML by the UML asynchronous operation call. However, this representation is *unfaithful* to mechanisms that notify communication failures.

### 5.5 Representing multicast message passing

UML has no single concept to represent multicast message passing. Therefore, to represent the multicast mechanism, we must introduce an intermediary object that deals with the pushing and pulling of messages to and from the publishers and subscribers.

We claim that this representation limits *platform independence*, because it forces the designer to make implementation choices regarding the implementation of the multicast mechanism. These choices include that the mechanism is implemented using a centralized intermediary object and threading mechanisms employed in the intermediary object.

Moreover, we argued that modelling a complete mechanism leads to *unsuitability* of a model.

### 5.6 Conclusion and discussion

We can conclude that UML can represent all the interaction mechanisms that we discussed. We consider UML suitable for representing synchronous request response, callback and one-to-one message passing, if the mapping from UML elements to interaction mechanisms that we explained above is used. We also consider UML platform independent with respect to synchronous request response and one-to-one message passing in CORBA and Web Services, because it maps equally well to both. We claim that UML is not platform independent with respect to callback, because it cannot distinguish a callback operation from two asynchronous operations. Hence, favouring a mapping onto two remote operation calls rather than on the callback mechanisms provided by the middleware. Also, UML requires the server to be aware that it is processing a callback, while CORBA does not require this. Also, we claim that UML is unfaithful with respect to representing system exceptions, because it does not consider them. One could argue that system exceptions have to be considered at a lower level of abstraction than the level at which UML should be used. However, this means that at the level at which UML is used, communication failure is not detected, making UML unsuitable for modelling reliable systems.

One can choose to increase the faithfulness of the UML elements, by explicitly modelling the mechanisms to detect communication failure. However, this would limit the level of suitability (because it means that more modelling elements must be used to represent each interaction) and platform independence (because the mechanism to detect communication failure represents design choice).

We consider UML unsuitable for representing polling and one-to-many message passing, because the interaction mechanisms to perform polling and one-to-many message passing must be modelled explicitly in UML; UML has no model elements that can be used to represent such interactions. Since modelling the mechanisms means revealing implementation details, this also restricts the level of platform independence. Table 1 summarizes our conclusions.

### 5.7 Example

Figure 9 illustrates our argument with a simple example. The example shows a chat application, which allows a user

**Table 1** Adequacy of UML for representing interaction mechanisms

| | Expressiveness | Suitability | Platform independence | Faithfulness |
|---|---|---|---|---|
| Synch. req/rsp | + | + | + | − |
| Callback | + | + | − | − |
| Polling | + | − | − | + |
| 1-to-1 messaging | + | + | + | − |
| 1-to-*n* messaging | + | − | − | + |

to join a conversation, then send and receive messages and leave the conversation again. When a communication failure occurs, the user is notified and the client exits. A publish/subscribe mechanism is used to implement communication between clients.

The example illustrates two problems with the adequacy of UML. Firstly, should UML provide concepts to represent multicast interaction, Fig. 9a would suffice as a specification. However, UML does not provide multicast concepts. Therefore, the modeller must include a specification of the multicast mechanism, like the one shown in Fig. 9b. Therewith the modeller reveals unnecessary implementation detail hence making the specification less platform independent and less suitable. Secondly, UML does not consider communication exceptions. Therefore, although Fig. 9a models a communication exception ('commError'), this event has no special semantics and can occur at any time, even if no communication action was taken before. Hence, the model is not faithful to any implementation.

## 6 Related work

This paper shows a technique to evaluate the adequacy of existing concepts for interaction design, based on how well they can represent interaction patterns. To the best of our knowledge such a technique does not yet exist in this area. However, similar techniques do exist in the area of workflow concepts (van der Aalst et al. 2003; van der Aalst and ter Hofstede 2002) and service description concepts (Wohed et al. 2003). With the evaluation of service description concepts being closest to our, since service description also involves interaction design.

There is a long history of research towards concepts for interaction design at various stages in a design process. Interaction design concepts have been studied in the area of reference models (such as ITU-T 1995, 1999), design languages (such as Bastide et al. 2000; ITU-T 2002; OMG 2003, 2004; Quartel et al. 2002; van Sinderen et al. 1992) and architectural description languages (such as Allan and Garlan 1997; Luckham and Vera 1995). The work presented in this paper is not meant to be another set of interaction design concepts. Rather, we present criteria to evaluate already existing concepts and we use these criteria to propose changes to existing concepts. These criteria form the contribution of our work. Although criteria for 'good' design concepts were implicitly used, and in some cases even made explicit, a thorough evaluation of these criteria has not been attempted before.

More recently, interaction patterns are being studied (Barros et al. 2005; Hohpe and Woolf 2004; Ruh et al. 2001). We use these interaction patterns to evaluate design concepts. In addition to that, we present interaction patterns

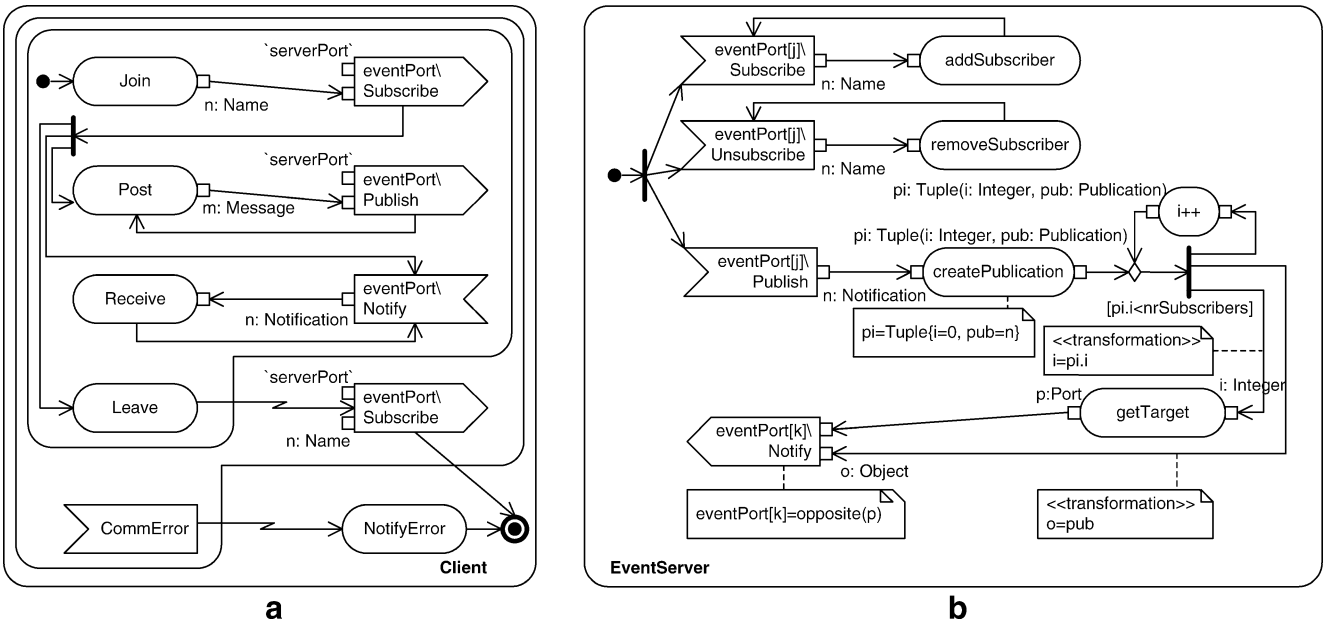



**Fig. 9** Example UML behavior models. **a** Client behaviour. **b** Event server behaviour

that are directly derived from existing middleware implementations of interaction mechanisms.

## 7 Conclusions and future work

In this paper we provided criteria for adequate interaction design concepts. We also provided an in depth analysis of the interaction mechanisms implemented by Web Services and CORBA and the UML 2.0 model elements for representing these interaction mechanisms.

Based on the criteria and the analysis of UML and the interaction mechanisms, we evaluated the adequacy of UML for representing these mechanisms. From this evaluation, we conclude that UML is unfaithful with respect to the representation of communication failure and unsuitable for representing polling and message passing mechanisms. This means that UML is not adequate for the design of reliable systems and for systems that use interaction mechanisms other than synchronous request/response, callback and one-to-one message passing.

In the context of the project in which this work is embedded, we aim to define concepts that are adequate for representing advanced interaction mechanisms (such as one-to-many message passing, transactions and negotiation) and reliable systems. In this paper we focused on design at the lowest level of abstraction before choosing a particular middleware platform (Web Services or CORBA). In future work we will also consider higher levels of abstraction and middleware platforms and develop concepts for those levels. Also, we will develop concepts to represent other aspects of interaction mechanisms, such as: threading mechanisms and creation and destruction of bindings between communicating parties (e.g. event channel subscriptions).

## References


Allan, R., & Garlan, D. (1997). A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology, 6*(3), 213–249.

Andrade Almeida, J. P., Dijkman, R. M., van Sinderen, M. J., Quartel, D. A. C., & Ferreira Pires, L. (2006). Model driven design, refinement and transformation of abstract interactions. *International Journal of Cooperative Information Systems, 15*(4), 599–632.

Barros, A., Dumas, M., & ter Hofstede, A. H. M. (2005). *Service interaction patterns*. In Proc. of the 3rd International Conference on Business Process Management (pp. 236–251).

Bastide, R., Sy, O., & Palanque, P. (2000). A formal notation and tool for the engineering of CORBA systems. *Concurrency: Practice & Experience, 12*, 1379–1403.

Dijkman, R. M., Dirgahayu, T., & Quartel, D. A. C. (2006a). *Towards advanced interaction design concepts*. In: Proc. of EDOC 2006 (pp. 331–342).

Dijkman, R. M., Quartel, D. A. C., & van Sinderen, M. J. (2006b). *Consistency in multi-viewpoint architectural design of enterprise information systems* (BETA Working Paper WP-188). Eindhoven, The Netherlands: Eindhoven University of Technology.

Hohpe, G., & Woolf, B. (2004). *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Boston, MA, USA: Addison Wesley.

ITU-T (1995). *Open distributed processing reference model (specification 901.4)*. Geneva, Switzerland: ITU-T.

ITU-T (1999). *Information technology—open distributed processing reference model—enterprise language (specification 911)*. Geneva, Switzerland: ITU-T.

ITU-T (2002). *Specification and description language (specification Z.100)*. Geneva, Switzerland: ITU-T.

Lamport, L., Shostak, R., & Pease, M. (1982). The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems, 4*(3), 382–401.

Luckham, D. C., & Vera, J. (1995). An event-based architecture definition language. *IEEE Transactions on Software Engineering, 21*(9), 717–734.

OMG (2002a). *CORBA component model 3.0 (specification formal/2002-06-65)*. Needham, MA, USA: Object Management Group.

OMG (2002b). *Common object request broker architecture: Core specification, version 3.0 (specification formal/02-12-06)*. Needham, MA, USA: Object Management Group.

OMG (2003). *UML 2.0 infrastructure specification (specification ptc/03-09-15)*. Needham, MA, USA: Object Management Group.

OMG (2004). *UML 2.0 superstructure specification (specification ptc/04-10-02)*. Needham, MA, USA: Object Management Group.

Quartel, D. A. C., Dijkman, R. M., & van Sinderen, M. J. (2005). *Extending profiles with stereotypes for composite concepts*. In Proc. of MODELS 2007 (pp. 232–247).

Quartel, D. A. C., Ferreira Pires, L., & van Sinderen, M. J. (2002). On architectural support for behavior refinement in distributed systems design. *Journal of Integrated Design and Process Science, 6*.

Ruh, W. A., Maginnis, F. X., & Brown, W. J. (2001). *Enterprise application integration: A Wiley tech brief*. New York, NY, USA: Wiley.

van der Aalst, W. M. P., & ter Hofstede, A. H. M. (2002). *Workflow patterns: On the expressive power of (Petri-net-based) workflow languages*. In Proc. of CPN 2002 (pp. 1–20).

van der Aalst, W. M. P., ter Hofstede, A. H. M., Kiepuszewski, B., & Barros, A. P. (2003). Workflow patterns. *Distributed and Parallel Databases, 14*(1), 5–51.

van Sinderen, M. J., Ferreira Pires, L., & Vissers, C. A. (1992). Protocol design and implementation using formal methods. *The Computer Journal, 35*, 478–491.

W3C (2004). *Web services architecture (specification NOTE-ws-arch-20040211)*. Sophia-Antipolis Cedex, France: W3C.

Wohed, P., van der Aalst, W. M. P., Dumas, M., & ter Hofstede, A. H. M. (2003). *Analysis of web services composition languages: The case of BPEL4WS*. In Proc. of the 22nd International Conference on Conceptual Modelling.

**Remco M. Dijkman** is an Assistant Professor at Eindhoven University of Technology in The Netherlands. He specializes in design methods and modelling techniques for business processes and enterprise information systems. He has written several scientific publications and served on program committees and organizing committees of several conferences. Previously he has been a consultant at Ordina, a Dutch IT consulting firm. He holds a Ph.D. degree and a Master's degree from the University of Twente in The Netherlands.

**Teduh Dirgahayu** is a Ph.D. candidate in Computer Science at the University of Twente. He has a Master's degree in Telematics from the University of Twente in The Netherlands. His research interests include design methods, architectures for distributed systems, service-oriented design, and model-driven engineering.

**Dick A.C. Quartel** is an Assistant Professor at the University of Twente, The Netherlands. His research interests include design methods and service architectures for networked systems, business process and service modelling, and semantic service interoperability. He is currently a Workpackage Leader in the Dutch Freeband A-MUSE project (BSIK 03025) on service design and semantic interoperability. He received his Master's and Ph.D. degrees in Computer Science from the University of Twente, The Netherlands.