

# Towards Advanced Interaction Design Concepts

Remco Dijkman<sup>1,2</sup>, Teduh Dirgahayu<sup>2</sup>, Dick Quartel<sup>2</sup>

<sup>1</sup>*Eindhoven University of Technology  
Eindhoven, The Netherlands  
r.m.dijkman@tm.tue.nl*

<sup>2</sup>*University of Twente  
Enschede, The Netherlands  
{t.dirgahayu, d.a.c.quartel}@utwente.nl*

## Abstract

*In this paper we analyse the interaction mechanisms provided by Web Services technology and by CORBA. Specifically we analyse the request/response, callback, polling and (multicast) message passing mechanisms. As a result we present Coloured Petri Nets that capture the behaviour of these mechanisms precisely.*

*Based on our analysis we define concepts for representing the Web Services and CORBA interactions in a suitable and platform independent manner. These concepts can be used for platform independent design of distributed applications, while they (provably) maintain the consistency with platform specific implementations. Because their behaviour is defined by Petri Nets, the concepts also support simulation, validation and verification of designs.*

*We also evaluate the suitability of UML's concepts for representing the mechanisms and the degree of platform independence that these concepts can achieve.*

## 1. Introduction

Middleware 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 mechanisms. Such design concepts help to:

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

A requirement on such design concepts is that they properly reflect the relevant properties of the represented

interaction mechanisms. This allows one to analyse the designs of systems that use these interaction mechanisms, assuming analysis techniques (e.g. validation, verification and simulation techniques) are defined for these concepts.

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. For practical purposes, the concept could be used for designs in which reliability is not an issue (such as the design of certain web-applications), but it can not be used in designs in which reliability is an issue (such as the design of a banking system).

The goal of our work is to provide concepts that properly represent interaction mechanisms. As a first step towards this goal, we analyse interaction mechanisms implemented in existing middleware, in particular CORBA [19] and Web Services [26]. To capture their properties precisely, we define these interaction mechanism using Coloured Petri Nets. Subsequently, we provide an initial set of concepts that properly represent the interaction mechanisms. These concepts must meet three criteria: expressiveness, suitability and platform independence. By *expressiveness* we mean that all relevant properties of the interaction mechanisms can be expressed. By *suitability* we mean the ease with which the interaction mechanisms can be expressed, preferably using a single concept or a small composition of concepts. For example, Petri Nets are sufficiently expressive, but are not suitable to express the interaction mechanisms considered in this paper in an easy and intuitive way. By *platform independence* we mean that the concepts do not imply implementation on a certain middleware.

An hypothesis underlying this paper is that existing interaction mechanisms are not properly represented by concepts in existing modelling languages. As a first test of this hypothesis, we analyse the concepts that UML 2.0 [17,18] provides to represent the interaction mechanisms we consider.

This paper is further structured as follows. Section 2 discusses the background of the research presented in this paper and introduces the notion of platform independence. Sections 3 and 4 analyse interaction mechanisms in Web Services and CORBA respectively, and model them in terms of Coloured Petri Nets. Section 5 proposes concepts that represent these mechanisms in a suitable and platform independent manner. Section 6 evaluates if and how the interaction mechanisms can be represented using UML 2.0. Section 7 presents related work. And section 8 discusses our conclusions and future work.

## 2. Advanced interaction concepts in MDA

The research presented in this paper is part of a larger research project. In this project we develop concepts for advanced interaction design in the context of a Model Driven Architecture (MDA) [16] design process.

An MDA process distinguishes platform independent models, which can be used to represent application functionality independent of the (middleware) platform on which the application will be implemented, and platform specific models. The main benefit of this approach is that it supports re-use of models by allowing the same platform independent model to be implemented on different platforms. 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 [2].

During a design process, interactions are considered at different levels of abstraction. For example, when modelling a business negotiation one may at first only consider the possible outcomes by defining the negotiation constraints of the involved parties. In a next step, one may define how the negotiation is performed, e.g., by successively breaking down multi-party negotiations into compositions of two party negotiations, and two party negotiations into patterns of negotiation offers, accepts and rejections. Subsequently, one could map these negotiations onto a transaction model to deal with unreliable parties or communication, and implement this model using CORBA interaction mechanisms.

This example shows the need to describe interactions at different abstraction levels, which correspond to different levels of platform independence. The availability of concepts to model such interactions would largely facilitate the platform independent modelling of interacting systems. Furthermore, the availability of transformations between interaction concepts used at different levels would facilitate the model-driven development of these systems. The goal of our project is to develop such concepts and transformations.

We focus on interaction concepts that are sufficiently generic to be used across different application domains.

We aim to derive these concepts from existing modelling languages, and by analysing frequently used interaction patterns in business processes, application designs and their implementations.

In this paper we start ‘from the bottom’ by analysing interaction mechanisms implemented in existing middleware to develop concepts that are suitable abstractions of these mechanisms. Our goal is to define concepts that can represent the Web Services and CORBA interactions, such that:

- the concepts do not reveal whether the interaction will be implemented using CORBA or Web Services;
- the observable behaviour of the concepts is equivalent to the observable behaviour of the corresponding Web Services and CORBA interactions.

We say that concepts that meet these criteria are *platform independent* (with respect to the Web Services and CORBA platforms).

## 3. Web services interaction mechanisms

Web Services technology provides various mechanism for interaction between software entities. We focus on the following mechanisms:

- one-to-one message passing communication;
- synchronous request/response communication;
- asynchronous request/response communication, based on callback (WS-Callback [5]);
- asynchronous request/response communication, based on polling (WS-Polling [25]);
- multicast message passing, based on publish/subscribe (WS-Notification [11]).

We represent the externally observable behaviour of each of these mechanisms, as it is observed by the entities that use them to interact with each other, abstracting from their bindings to concrete protocols.

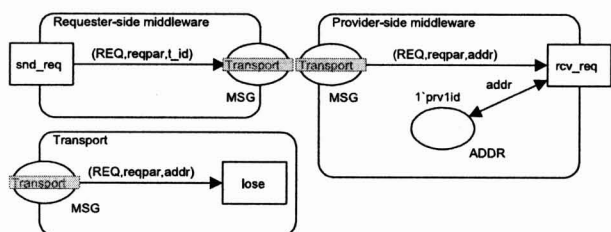
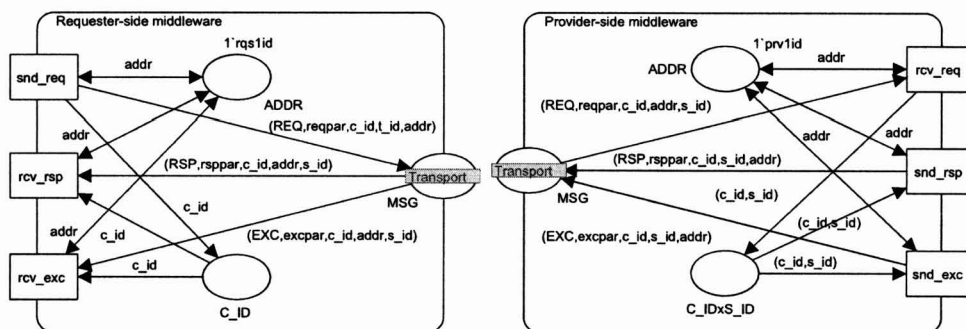


Figure 1. Web services message passing

### 3.1. One-to-one message passing

In one-to-one message passing mechanisms, we distinguish a *service requester*, which sends a request, a *service provider*, which receives a request and performs the service being requested. Web Services technology implements this mechanism using a one-way operation.



**Figure 2. Web services synchronous request/response**

Figure 1 presents a Coloured Petri Net that represents the observable behaviour of a one-way message passing mechanism. The 'snd\_req' transition represents the sending of a request at the requester-side. The 'rcv\_req' transition represents the reception of a request at the provider-side. 'REQPAR' represents the request parameter that the designer can specify freely. 'ADDR' is an abstract type that represents the address at which the server is located.

At the requester-side, a requester sends a request with the address of the server ('t\_id'). At the provider-side, the request is received by the service provider. The address at which the request is received must match the provider's address, which is stored on the place with the initial marking '1'prv1id'. The transport layer transports the request from the requester-side to the provider-side, but may lose the request because of communication failure.

### 3.2. Synchronous request/response

In synchronous request/response communication, the service requester waits for a response or exception message from the service provider after sending a request. Web Services technology implements this mechanism using a request-response operation.

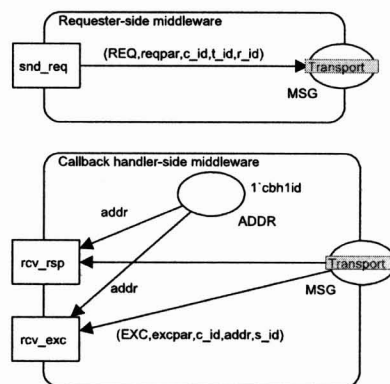
Figure 2 presents a Coloured Petri Net that represents the externally observable behaviour of a synchronous request/response mechanism. The 'rcv\_rsp' and 'rcv\_exc' transitions represent the reception of a response and exception at the client-side, respectively. The 'snd\_rsp' and 'snd\_exc' transitions represent the sending of a response and user exception at the server side, respectively. 'RSPPAR' and 'EXCPAR' represent response and exception parameters that the designer can specify freely. 'C\_ID' is an abstract type that uniquely identifies a request/response invocation.

At the requester-side, the requester adds its own address ('addr') and an id ('c\_id') to the request message, so that the provider knows where to direct the corresponding response or exception. At the provider-

side, after receiving a request message, the provider sends a response or exception in return.

### 3.3. Asynchronous request/response: callback

WS-Callback provides an asynchronous request/response mechanism without discriminating whether a response is a normal response or user exception. The response is to be handled by the requester itself or by another service. Figure 3 presents a Coloured Petri Net that represents the observable behaviour of the callback mechanism.



**Figure 3. Web services callback**

At the requester-side, the requester sends the service provider a one-way request with which it passes an address ('r\_id') at which it expects the response.

At the provider-side, after receiving a request, the provider sends a response as usual. However, because the requester passes the address of the callback handler as the address to which the response will be sent, the response arrives at the callback handler. Also, instead of sending a response or an exception as part of the operation in which the request was defined, the callback handler provides an operation for receiving a response ('rcv\_rsp'). If the callback handler expects that an exception may arrive, it

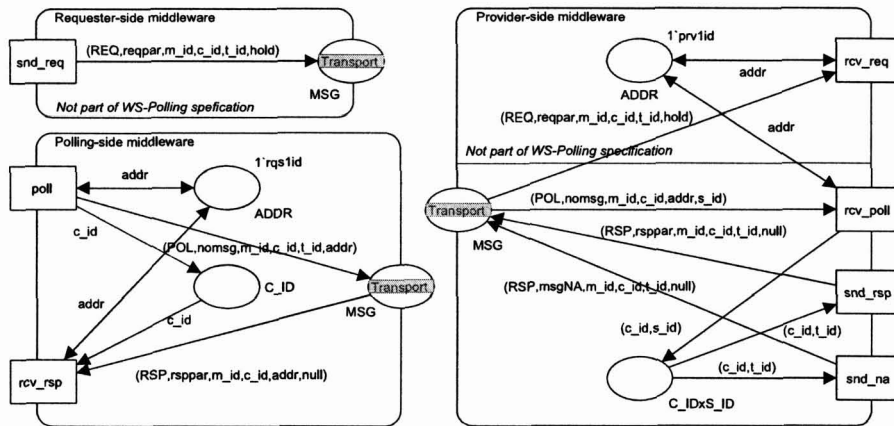


Figure 4. Web services polling

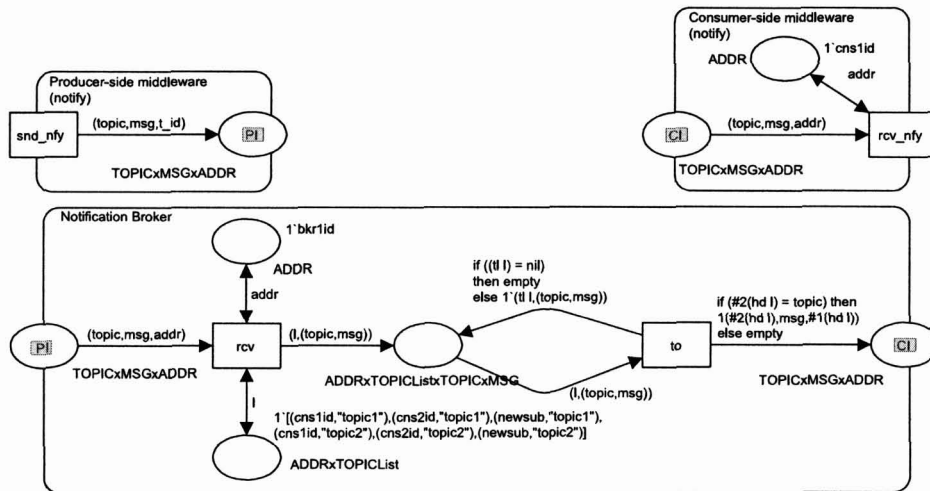


Figure 5. Web services multicast message passing

must also provide an operation for receiving the exception ('rcv\_exc').

### 3.4. Asynchronous request/response: polling

WS-Polling specifies how a service retrieves a response from a service provider. The poll must be associated with a message identifier ('m\_id') that correlates the response to a preceding request (with the same identifier).

Figure 4 presents a Coloured Petri Net that represent the observable behaviour of the polling mechanism. At the requester-side, the requester sends a poll request to the provider at which a response is expected to be available. The poll request carries message id ('m\_id') of the request for which the response must be retrieved.

At the provider-side, the provider can return: the response desired by the requester ('snd\_rsp') or a message indicating that the desired response is not available yet ('snd\_na').

### 3.5. Multicast message passing

In multicast message passing, one party can send messages to many others.

Web Services technology provides a multicast message passing mechanism through the WS-notification service, which is based on the publish/subscribe mechanism. In such a mechanism, a producer publishes notification messages ('NMSG') on a certain topic ('TOPIC'). These messages are obtained by all consumers that are subscribed to the topic. We focus on the mechanism that uses a 'broker' as an intermediary between the producers and the consumers. Producers can send notification messages to the broker ('snd\_nfy'), which the broker then passes to consumers ('rcv\_nfy').

Figure 5 represents the behaviour of the notification producers and consumers and of the broker that is the intermediary between them. A producer makes a message available to the broker by putting it on the place denoted

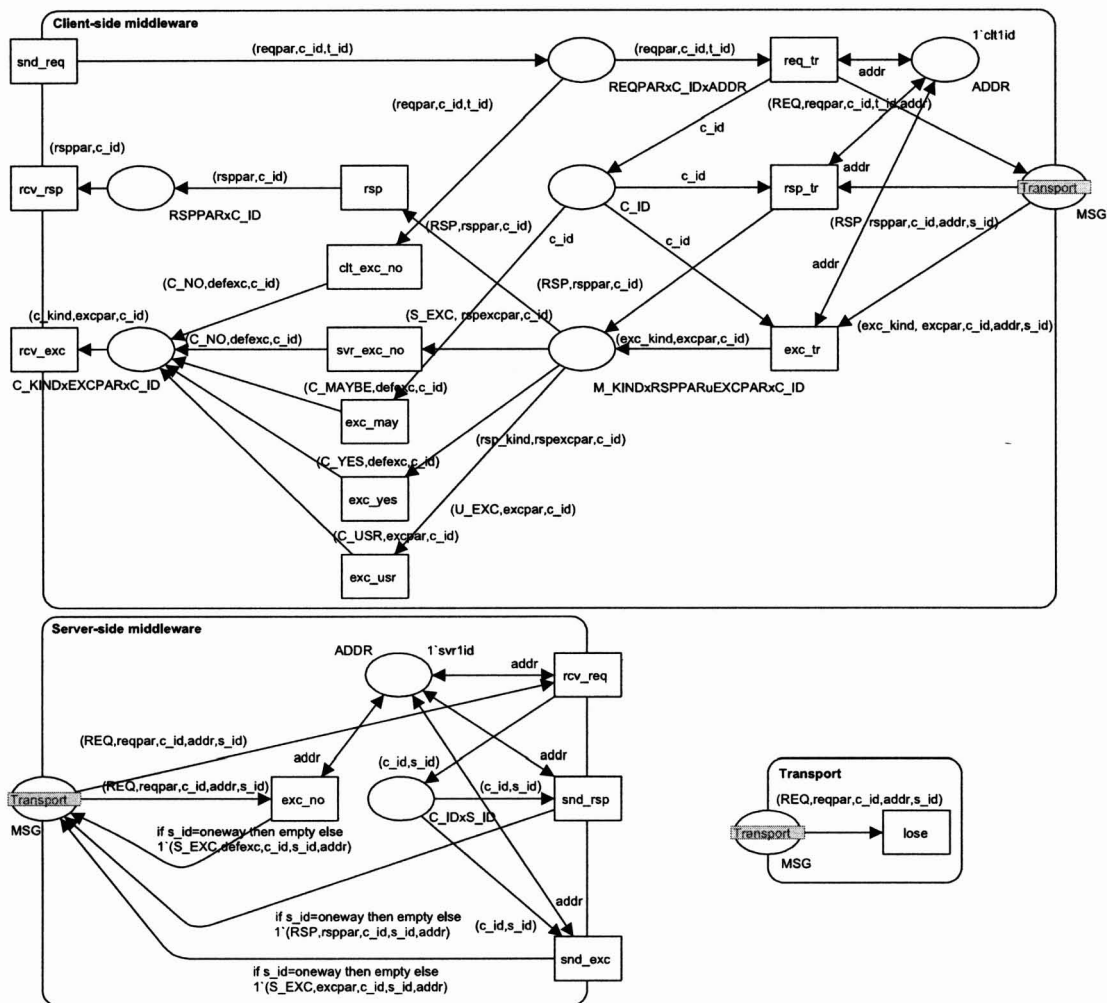


Figure 6. CORBA request/response

as ‘PI’ (for producer interface). The broker maintains a list of consumer subscriptions to topics. This list is on the place coloured ‘ADDRxTOPICList’. The broker then produces a message for each consumer that is subscribed to the topic of the producer’s message. It puts these messages on the place denoted as ‘CI’ (for consumer interface), where the appropriate consumers can obtain it.

#### 4. 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;

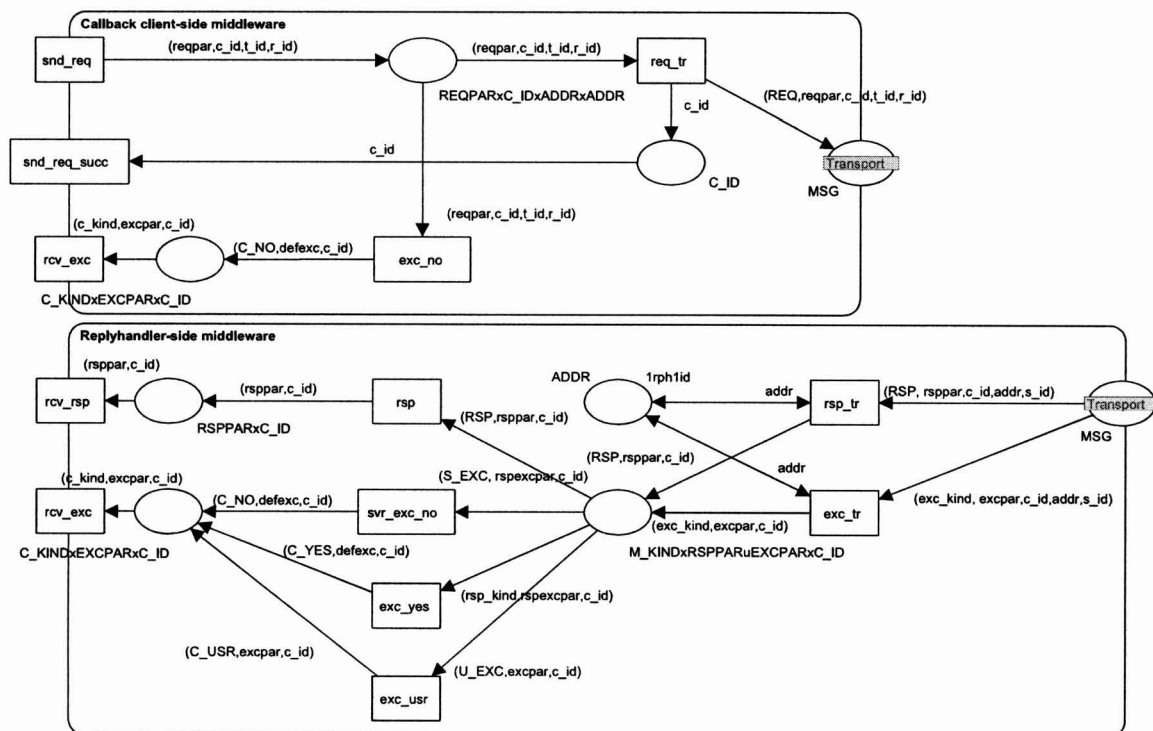
- one-to-one message passing (also called oneway operations in CORBA);
- multicast message passing, based on publish/subscribe (also called the Notification Service in CORBA).

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.

##### 4.1. 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 6 presents a Coloured Petri Net that represents the observable behaviour of a synchronous request/response mechanism. 'C\_KIND' represents the kind of exception 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/reponse 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 figure 6 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*” [19] (page 22-55)).

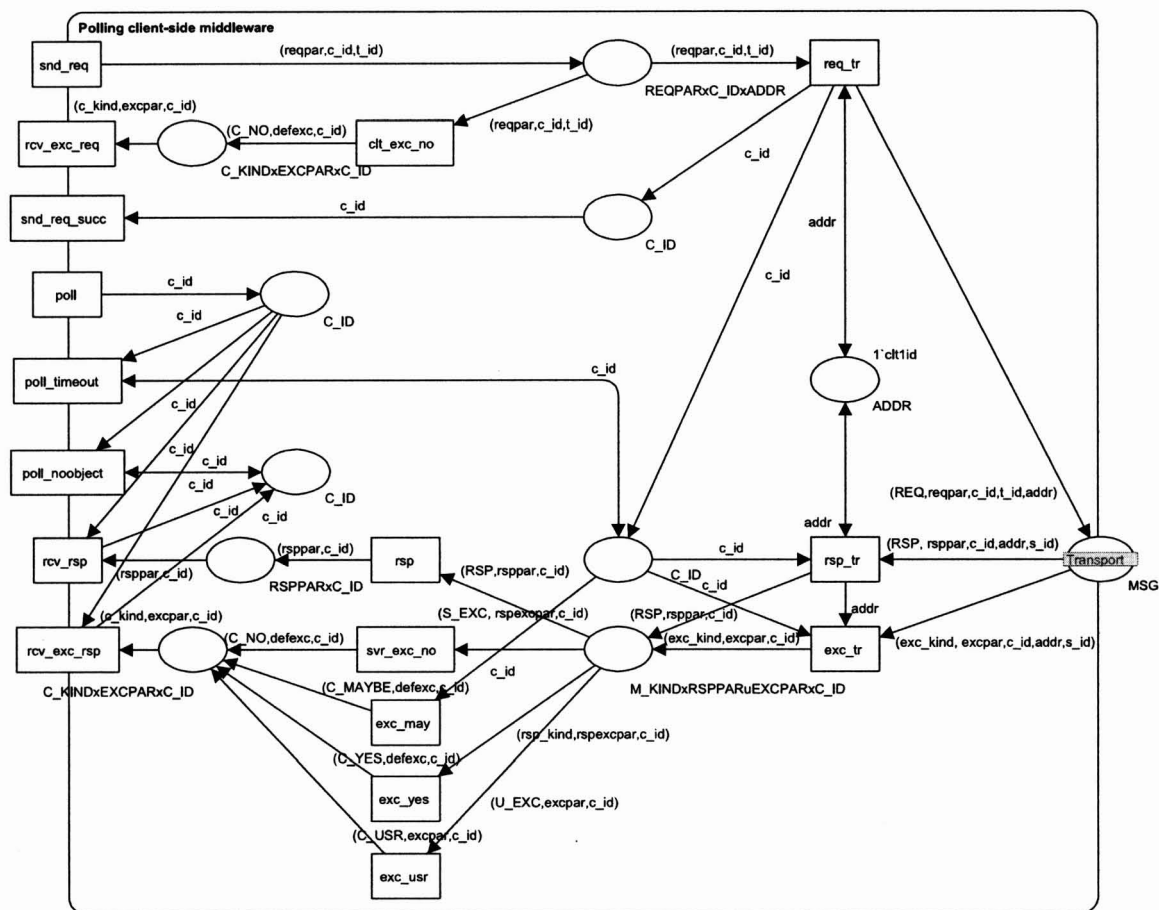


Figure 8. CORBA polling

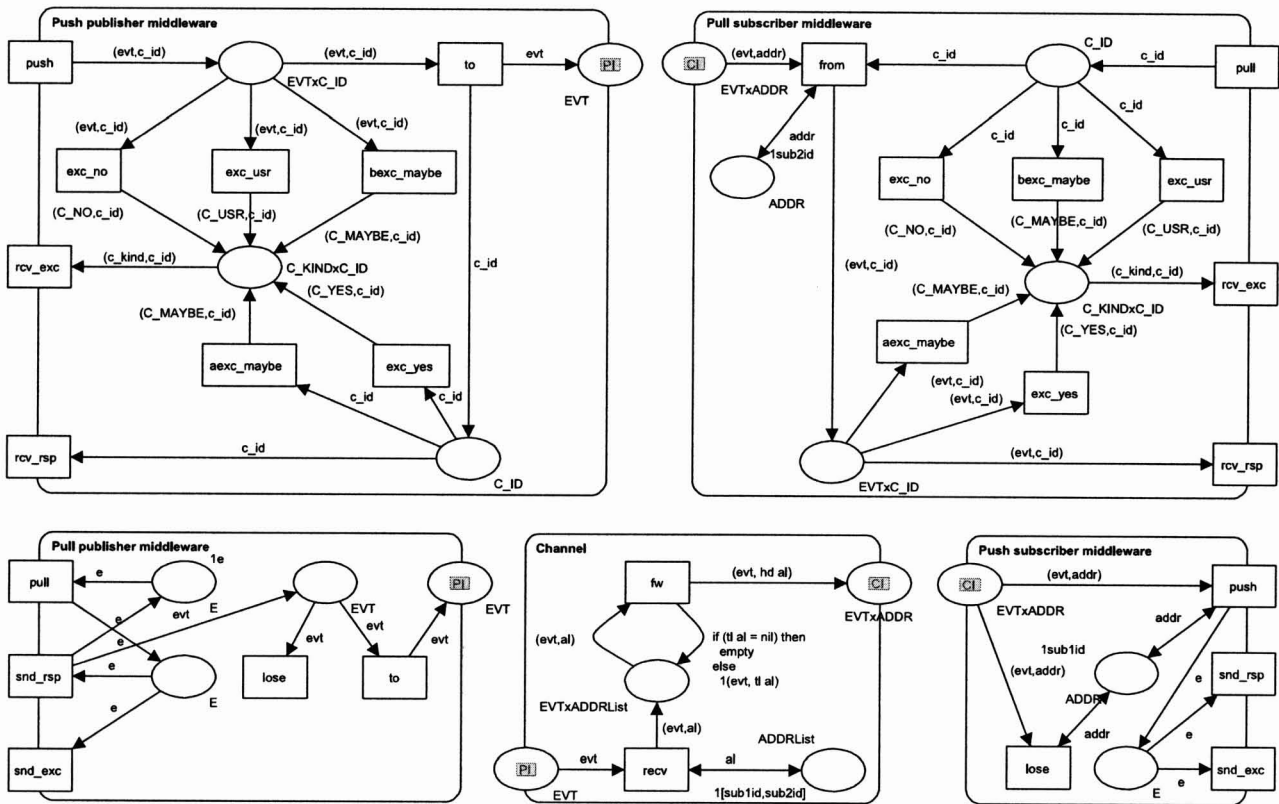
#### 4.2. Asynchronous request/response: callback

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. Figure 7 presents a Coloured Petri Net that represent the observable behaviour at the client-side and the reply-handler side. The Petri Net is similar to the Petri Net from figure 6, 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 figure 7 and the 'server-side' part from figure 6 to the same 'transport'.

#### 4.3. 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 8 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 ('rcv\_rsp') carrying



**Figure 9. CORBA multicast message passing**

the response; or a notification ('rcv\_exc\_rsp') carrying an exception.

#### 4.4. 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 'oneway' request/response communication. For this mechanism, nothing changes in the server represented in figure 6. However, the server will not send a response upon receiving a request for 'oneway' communication. At the client side, mechanisms for dealing with responses and exceptions that occur after the request was sent, can be removed.

#### 4.5. Multicast message passing

Like Web Services technology, CORBA implements multicast message passing using a publish/subscribe mechanism. However, where Web Services technology only supports a 'push' strategy, CORBA supports both a 'push' and a 'pull' strategy, such that publishers can either 'push' messages to the 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 9 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').



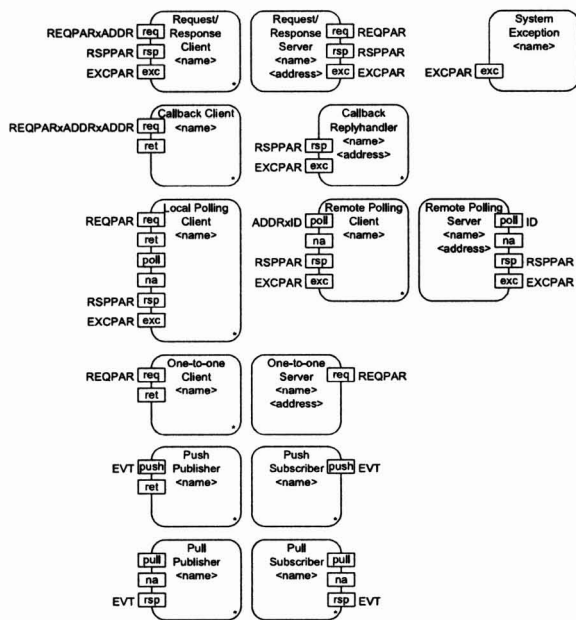


Figure 10. Proposed concepts

## 5. Towards representing interactions in a platform independent manner

In this section we present an initial set of concepts to represent interaction mechanisms in a distributed application design. In a design, instances of our interaction concepts must be combined with instances of concepts from some behaviour formalism.

We claim that our concepts meet the criteria for platform independence, namely that they:

- do not imply implementation of a model onto a particular platform (currently we only consider Web Services and CORBA);
- have the same externally observable behaviour as the interaction mechanisms to which they will be transformed.

However, we still plan to define the behaviour of the concepts using Petri Nets, such that we can formally verify the second point. This possibly leads to a revised version of the proposed concepts.

Figure 10 shows the concepts that we propose for platform independent modelling. The proposed graphical notation is tentative. It resembles the representation of the Petri Nets from previous sections, to illustrate how the concepts can be formalized using Petri Nets.

The synchronous request/response mechanism is separated into a client and a server part. The client part represents interactions between the client and the platform that provides the request/response mechanism. It accepts requests from the client and notifies responses or user

exceptions to the client. Similarly, the server part represents interaction between the server and the platform. It notifies requests to the server and accepts responses or user exceptions from the server. The client and the server part are parameterised with the name of the request/response pair. The server part is also parameterised with an address. The client can address requests to that address. Exceptions notified to the client are only user exceptions. System exceptions are notified to the client by the 'System Exception' part. In this way, user and system exception concerns are separated.

The client and server parts can be directly transformed into a Web Services implementation and the client, server and system exception parts can be directly transformed into a CORBA implementation. For a Web Services implementation, an implementation of the system exception part must be provided by the designer.

For a callback, we only change the client part, such that it is split-up into a client and a replyhandler. The client and replyhandler parts can be associated with a regular request/response server part. The transformation onto both CORBA and Web Services is straightforward. However, the Web Services implementation assumes that the server maintains a reference to the replyhandler to which the response will be sent.

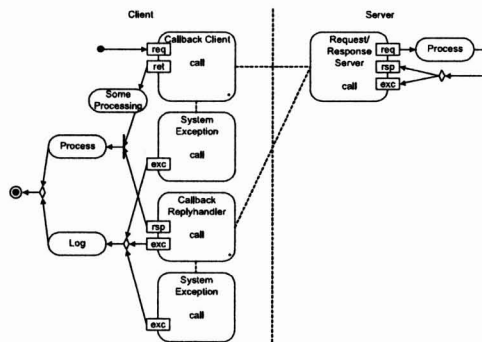
We represent two different polling mechanisms. With the local polling mechanism the server operates normally, but the client side middleware locally stores the response, which it provides to the client when polled. The 'na' interaction represents that no response is available yet. This mechanism can be directly transformed into a CORBA implementation. The transformation into a Web Services implementation is more complex, because the implementation of the client side middleware will have to be generated. With the remote polling mechanism the client will directly poll the (remote) server for the response, which the server will provide. This mechanism can be directly transformed into a Web Services implementation. For the CORBA implementation, it will have to be transformed into a regular request/response, with the added property that it polls for a response of a previous request.

We represent one-to-one message passing by a client part that can send requests and a server part that can receive them. The transformation of the one-to-one message passing mechanism is straightforward for both a CORBA and a Web Services implementation.

We represent (publish/subscribe) multicast message passing, using the push publisher, pull publisher, push subscriber and pull subscriber concepts. The concepts are parameterized with a name. The name can represent the name of a channel. However, in case messages have a topic, the name can represent the name of channel and the name of topic. In this way, if, for example, subscribers are

subscribed to two different topics on the same channel, we can represent this by two instances of a subscriber concept; one for each topic to which the subscriber is subscribed.

The concepts marked ‘\*’ can be associated with a system exception handler. In case of a pull publisher and push subscriber the system exception handler can be used by the publisher or subscriber to notify the channel that it is not connected.



**Figure 11. Proposed concepts**

Figure 11 shows a simple example in which the concepts from this section are used in a UML activity diagram. We can integrate the concepts with UML activity diagram concepts if we consider that both are defined on a 'token game' semantics. Interaction concept instances that belong to the same interaction are connected by a dashed line. Tokens flow between concept instances connected by the dashed lines, as defined by the semantics of the concepts. The example illustrates a client that has a request/response interaction called 'call' with a server and expects the result in a callback. The client sends the request, after which it can do some processing. The server processes the request and either returns a response or an exception. If the client receives an exception or a system exception, it logs the exception and ends the activity. If the client receives a response, it processes the response.

## 6. Representing interactions in UML

In this section we discuss the suitability of UML for representing the interaction mechanisms analysed above.

### 6.1. Representing request/response

We represent request/response interactions, using the UML synchronous operation call. We can do that provided that we choose that UML requests and responses are sent through an unreliable medium that does not preserve ordering.

A UML operation does not support the representation of system exceptions. Hence, it cannot be used to

represent a system in which we consider system exceptions, such as exceptions notified when a request cannot be sent or when a request or response is lost. It also leaves the client unable to detect whether a request or response was lost, because no exception is generated. We can solve this by modelling a timeout mechanism that detects [21] message loss after a certain timeout. However, this illustrates the limited *suitability* of UML for representing request/response mechanisms that may fail.

The limited suitability of UML for representing system exceptions may not be considered a problem when representing Web Services interactions, because the Web Services specification does not mention system exceptions. However, a designer will eventually also want to consider system exceptions in systems implemented on Web Services technology.

### 6.2. 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. To distinguish this construct from two regular UML operations, we must stereotype the operations.

We claim that this representation is too implementation oriented, because: (i) exceptions cannot be declared as such, but have to be specified as asynchronous operation calls; and (ii) it requires the server-side to be aware that it is being called using a callback operation, while we learned from the CORBA implementation that this is not necessary.

### 6.3. 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. To distinguish this construct from a regular UML operation, we must stereotype the operation.

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

1. accepts a synchronous 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; and
5. accepts (synchronous) polling calls from the client and returns the response if it is available.

We claim that, similar to the callback mechanism, this representation is too implementation oriented. 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.

#### **6.4. Representing one-to-one message passing**

We can represent one-to-one message passing in UML by the UML asynchronous operation call.

#### **6.5. 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 is too implementation oriented, 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.

#### **6.6. Conclusion**

We conclude that UML has limited suitability to represent system exceptions. Because to represent system exceptions, we must represent the mechanisms that generate system exceptions, such as time-out detection.

Also, UML forces the designer to make choices regarding the implementation of callback, local polling and multicast message passing. Hence, it limits the level of platform independence that can be achieved.

### **7. Related work**

There is a long history of research towards concepts to represent interactions at various stages in a design process. Interaction concepts have been studied in the area of reference models (such as [13,14]), design languages (such as [6,7,9,12,17,18,22,24]) and architectural description languages (such as [1,15]). Our work aims to contribute to these areas by evaluating and improving the concepts that are defined in these areas.

More recently, interaction patterns are being studied [3,10,23]. Interaction patterns represent frequently occurring compositions of interactions, mainly in the context of business interaction. The project in which the work presented in this paper is embedded, aims to contribute to this area by defining the implementation relation between these business interactions and interactions that can support them in existing middleware.

Platform specific languages (typically UML profiles) exist to graphically represent Web Services and CORBA interactions. The CORBA profile for UML [20] is an example of such a language. Our work complements this work, because we do not aim to define a graphical notation to represent interactions, but to define concepts that define the behaviour of those interactions precisely. Languages that represent the behaviour of Web Services or CORBA based applications include [4,8]. However, this work focuses on the behaviour in which interactions are used, while we focus on the behaviour of the interactions themselves.

### **8. Conclusions and future work**

In this paper we provided an in depth analysis of the interaction mechanisms implemented by Web Services and CORBA. Based on this analysis, we defined concepts that can represent the mechanisms in a suitable and platform independent manner. Furthermore, we showed how we can prove that these concepts have the same externally observable behaviour as the interaction mechanisms by which they will be implemented. The actual proof is left for future work.

Also, we evaluated the suitability of UML for representing these mechanisms. Based on this evaluation, we argue that UML can only achieve limited suitability and platform independence when representing system exceptions and callback, polling and message passing mechanisms.

In the context of the project in which this work is embedded, we aim to add more concepts and/or elaborate on the concepts that we defined in this paper. 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).

## Acknowledgements

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.

## 9. References

- [1] R. Allan, and D. Garlan, "A Formal Basis for Architectural Connection," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, pp. 213-249, 1997.
- [2] J.P. Andrade Almeida, R.M. Dijkman, M.J. van Sinderen, D.A.C. Quartel, and L. Ferreira Pires. "Model Driven Design, Refinement and Transformation of Abstract Interactions," Accepted for publication in: *Int. J. of Cooperative Information Systems (IJCIS)*, 2006.
- [3] A. Barros, M. Dumas, and A.H.M. ter Hofstede, "Service Interaction Patterns," In: *Proc. of the 3rd International Conference on Business Process Management (BPM)*, pp. 236-251, 2005.
- [4] R. Bastide, O. Sy and P. Palanque, "A formal notation and tool for the engineering of CORBA systems," *Concurrency: Practice & Experience*, vol. 12, pp. 1379-1403, 2000.
- [5] BEA Systems, "WS-Callback Protocol (WS-Callback)," Available at: [http://dev2dev.bea.com/webservices/WS-Callback-0\\_9.html](http://dev2dev.bea.com/webservices/WS-Callback-0_9.html), 2003
- [6] T. Bolognesi, J. van de Lagemaat, and C.A. Vissers, *LOTOSphere: Software Development with LOTOS*. Dordrecht: Kluwer Academic Publishers, 1995.
- [7] R.J. van Glabbeek and U. Goltz, "Refinement of Actions and Equivalence Notions for Concurrent Systems," *Acta Informatica*, vol. 37, pp. 229-327, 2001.
- [8] T. Gardner, "UML Modelling of Automated Business Processes with a Mapping to BPEL4WS," Available at: <http://www.cs.ucl.ac.uk/staff/g.piccinelli/eoows/documents/paper-gardner.pdf>, 2003.
- [9] R. Gorrieri and A. Rensink, "Action Refinement," in *Handbook of Process Algebra*, J. A. Bergstra, A. Ponse, and S. A. Smolka, Eds.: Elsevier, 2001, pp. 1047-1147.
- [10] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*: Addison-Wesley, 2004.
- [11] IBM (ed.), "Web Services Notification," Available at: <http://www-128.ibm.com/developerworks/library/specification/ws-notification/>, 2004
- [12] ITU-T, "Specification and Description Language (SDL)," Specification Z.100 (08/02), 2002.
- [13] ITU-T and ISO/IEC, "Open Distributed Processing Reference Model (ODP-RM)," ITU-T Specification 901.4 and ISO/IEC Specification 10746-1.4, 1995.
- [14] ITU-T and ISO/IEC, "Information Technology - Open Distributed Processing Reference Model - Enterprise Language," ITU-T Specification 911 and ISO/IEC Specification 16414, 1999.
- [15] D.C. Luckham, and J. Vera, "An Event-Based Architecture Definition Language," *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 717-734, 1995.
- [16] Object Management Group, "MDA-Guide, V1.0.1," Specification omg/03-06-01, 2003.
- [17] Object Management Group, "UML 2.0 Infrastructure Specification," Specification ptc/03-09-15, 2003.
- [18] Object Management Group, "UML 2.0 Superstructure Specification," Specification ptc/04-10-02, 2004.
- [19] Object Management Group, "Common Object Request Broker Architecture: Core Specification, Version 3.0," Specification formal/02-12-06, 2002.
- [20] Object Management Group, "UML Profile for CORBA," Specification formal/02-04-01, 2002.
- [21] D.A.C. Quartel, R.M. Dijkman, and M.J. van Sinderen, "Extending Profiles with Stereotypes for Composite Concepts," In: *Proc. of the 8th ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS)*, 232-247, 2005.
- [22] D.A.C. Quartel, L. Ferreira Pires, and M.J. van Sinderen, "On Architectural Support for Behavior Refinement in Distributed Systems Design," *J. of Integrated Design and Process Science*, vol. 6, 2002.
- [23] W.A. Ruh, F.X. Maginnis, and W.J. Brown. "Enterprise Application Integration: A Wiley Tech Brief." John Wiley and Sons, Inc, 2001.
- [24] M.J. van Sinderen, L. Ferreira Pires, and C.A. Vissers, "Protocol design and implementation using formal methods," *The Computer J.*, vol. 35, pp. 478-491, 1992.
- [25] W3C, "WS-Polling," Specification SUBM-ws-polling-20051026, 2005.
- [26] W3C, "Web Services Architecture," Specification NOTE-ws-arch-20040211, 2004.