A Systematic Approach to Platform-Independent Design Based on the Service Concept

João Paulo Almeida, Marten van Sinderen, Luís Ferreira Pires, Dick Quartel
Centre for Telematics and Information Technology, University of Twente
PO Box 217, 7500 AE Enschede, The Netherlands
{almeida, sinderen, pires, quartel}@cs.utwente.nl

Abstract

This paper aims at demonstrating the benefits and importance of the service concept in the model-driven design of distributed applications. A service defines the observable behaviour of a system without constraining the system's internal structure. We argue that by specifying application-level interaction aspects as a service, and designing application parts in terms of this service, the design of application parts is not constrained by interaction patterns provided by a middleware platform. Therefore, a level of platform-independence can be achieved, so that the design of application parts can be reused across a large set of middleware platforms. The service concept is also used in our approach to describe an abstract platform that defines what characteristics of a potential target middleware platform are considered in platform-independent design. We discuss the trade-offs a designer is confronted with in the definition of an abstract platform, and discuss alternatives for platform-specific realization.

Keywords: platform-independence, middleware, Model Driven Architecture, service concept

1. Introduction

Model Driven Architecture (MDA) development is increasingly gaining support as an approach to manage system and software complexity in distributed application design [7]. MDA development focuses first on the functionality and behaviour of a distributed application, which results in platform-independent models (PIMs) of the application that abstract from the technologies and platforms that will be used to implement the application. Subsequent steps lead to a mapping from PIMs to a platform-specific implementation (PSI), possibly via platform-specific models (PSMs). The main advantages of MDA development – software stability, software quality and return on investment – stem from the possibility to derive different PSIs (via different PSMs) from the same PIMs, and to automate to some extent the model transformation process.

The concept of platform-independence plays a central role in MDA development. We believe that platform-independence can only be defined once a set of target platforms is known, such that their general capabilities and their irrelevant technological and engineering details can be established. This leads to the observation that there can be several PIMs, including various levels of PIMs, dependent on whether one wants to consider different sets of target platforms. Another observation is that different application characteristics or different sets of target platforms generally lead to different types of (intermediate) models, design structures or patterns, and model transformations.

The objective of this paper is to investigate what types of models can be useful in the MDA development trajectory, how these models are related, and which criteria should be used for their application. More specifically, we aim at demonstrating the benefits and importance of the service concept in a model-driven design trajectory. Since a service is a design that defines the observable behaviour of a system without constraining the system's internal structure, it is possible to describe systems without relying on support provided by a particular concrete middleware platform. In this respect, the service concept is particularly useful in the definition of application interaction aspects, and in the definition of general capabilities of middleware platforms. By using this approach to middleware application development, a level of platform-independence can be achieved, so that the design of application parts can be reused across a large set of middleware platforms.

This paper is further structured as follows: Section 2 presents the service concept; Section 3 discusses the notion of platform independence; Section 4 advocates the use of application interaction systems to capture platform-independent interaction aspects; Section 5 presents our proposed model-driven design trajectory, and Section 6 applies this design trajectory to an example. Finally, Section 7 presents our conclusions and outlines some future work.

2. The Service Concept

The Webster's dictionary provides a definition of system particularly applicable to distributed systems: *A system is a regularly interacting or interdependent group of items forming a unified whole*.

This definition indicates two different perspectives of a system: an integrated and a distributed perspective. The integrated perspective considers a system as a whole or black box. This perspective only defines what function a system performs for its environment. The distributed perspective defines how this function is performed by an internal structure in terms of system parts (which are also systems) and their relationships. Figure 1 depicts both system perspectives.

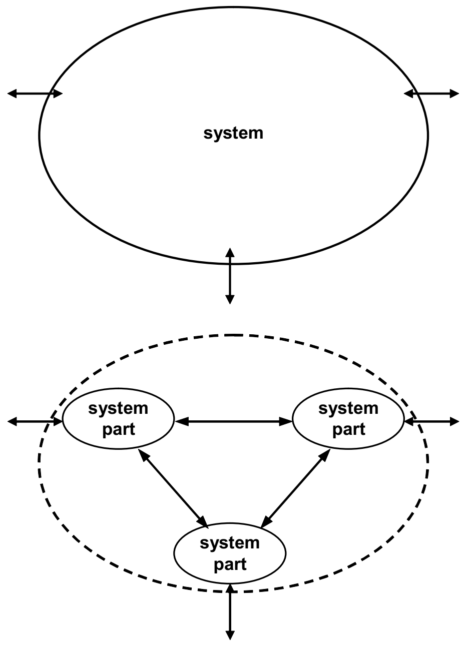

Figure 1. Integrated and distributed perspective of a system

We call the integrated perspective of a system a *service* [22]. A service is a design that defines the observable behaviour of a system in terms of the interactions that may occur at the interfaces between the system and the environment and the relationships between these interactions. A service does not disclose details of an internal organization that may be given to implementations of the system [23].

Since the concept of system is recursive, in the sense that a system part is a system in itself, the service concept can be applied recursively in a system. The recursive application of the service concept allows a designer to consider the behaviour of a system at different related decomposition levels. In general, the number of decomposition levels and the particular choices for decomposition depend on particular system requirements and objectives of a designer.

When interactions between system parts have to be explicitly designed, the concept of interaction system is introduced. An *interaction system* supports the set of related interactions between two or more systems parts [14, 16]. An interaction system consists of parts of system parts and their means of interaction, as depicted in Figure 2.

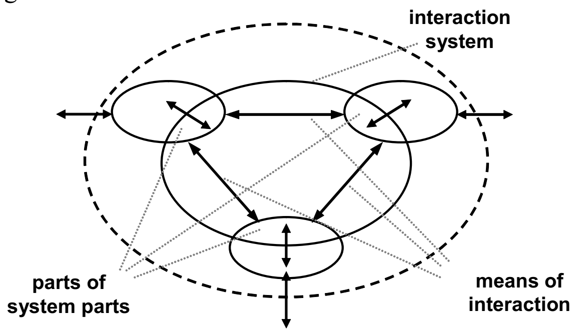

Figure 2. Interaction system from a distributed perspective

The complexity of interaction systems and thus the involvement of system part, varies, depending on the interactions that need to be considered. For example, when interactions concern application interworking, the interaction system will be more complex than when bit transfer is considered.

An interaction system is a system in itself, and therefore the behaviour of an interaction system can be defined as a service, as depicted in Figure 3. The service specification serves as a starting point for the design of an interaction system that supports the service.

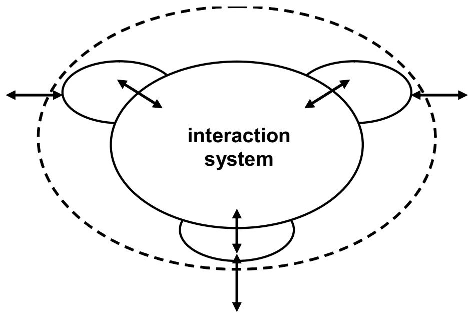

Figure 3. Interaction system from an integrated perspective

Interaction system design and the service concept play an important role in the design of protocol systems [22]. A systematic design method for protocols [21] consists of (i) defining the service to be supported by a service provider in terms of the service primitives that occur at service access points, and the relationships between service primitives; and, (ii) decomposing this service in terms of a structure of protocol entities and a lower level service. This resulting structure, which we call a *protocol*, has to be a correct implementation of the service. This can be assessed formally, if both the service and protocol are specified using some formal language.

Interaction systems that satisfy basic communication needs between software components have been referred to as *connectors* in the software architecture literature [1].

3. Platform-independence

3.1. Middleware-centred Development

In middleware-centred development, a system is structured in terms of a *middleware platform* and a collection of application parts, often called *objects* or *components.* A middleware platform provides a (limited) set of interaction patterns to support the interaction of application parts. There are several different types of middleware platforms, each one offering different types of interaction patterns. Examples of these patterns are *request/response*, *message passing* and *message queues*. Examples of middleware platforms are CORBA/CCM [9, 10], .NET [6], and Web Services [24, 25].

Design methods based on the re-use of middleware platforms often consist of partitioning the application into application parts and defining the interconnection aspects by defining interfaces between parts, e.g., by using object-oriented techniques and abstracting from distribution aspects. The available constructs to build interfaces are constrained by the interaction patterns supported by the targeted platform. Examples of these constructs are *operation invocation*, *event sources and sinks*, and *message queues*.

The predominance of this structuring strategy emphasizes a structuring of applications in terms of the choice of interaction patterns provided by a particular middleware platform. The design of the application is therefore platform-specific, not only in the sense that the design depends on particular technological conventions adopted by the middleware platform, but also in the sense that the structure of the application depends on the set of interaction patterns provided.

3.2. MDA approach

In order to shield the design of applications from the choice of platform and guarantee the re-use of designs across different platforms, the concept of platform-independence has been introduced in the MDA approach adopted by the Object Management Group (OMG) and others.

Platform-independence is a quality of a model that relates to the extent to which the model relies on characteristics of a particular platform. In this paper, we assume that models are used to specify both the behaviour and structure of a system or system part, and that several platform-independent models may be used in conjunction to specify a *design*. A consequence of the use of platform-independent models is the ability to refine the design or implement it on a number of target platforms.

The term *platform* is used to refer to technological and engineering details that are *irrelevant* to the fundamental functionality of a system (part) [8]. In order to refer to platform-independent or platform-specific models, one must define what a platform is, i.e., one must define which technological and engineering details are irrelevant *in a particular context*. For the purpose of this paper, we assume that a platform corresponds to some specific middleware technology.

Ideally one could strive for PIMs that are absolutely neutral with respect to all different classes of middleware technologies. However, we foresee that at a certain point in the development trajectory, different sets of platform-independent modelling concepts may be used, each of which is needed only with respect to specific classes of target middleware platforms. Figure 4 illustrates an MDA design trajectory, in which such a highly abstract and

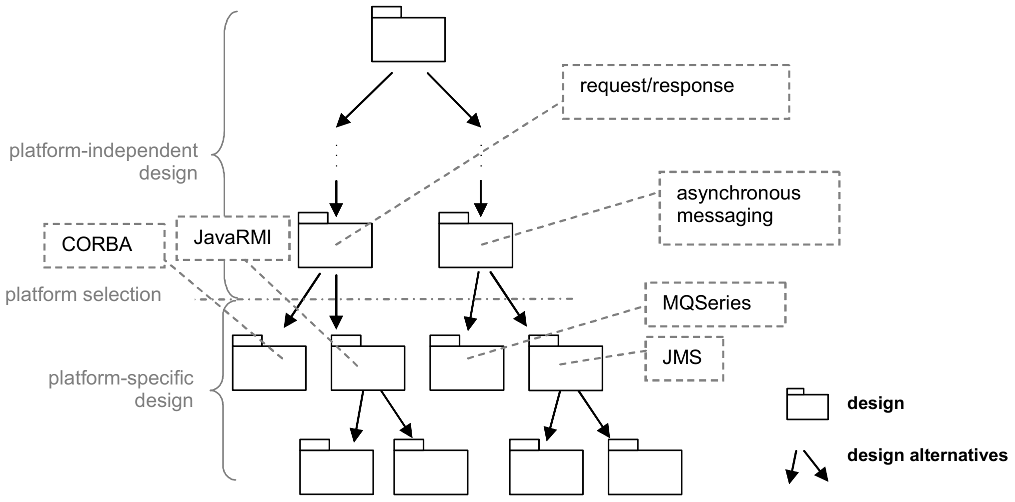

Figure 4. An MDA design trajectory

neutral PIM is depicted as the starting point of the trajectory. In Figure 4, the platform-independent models are defined that facilitate the transformation to two particular classes of middleware platforms, namely request/response (object-based) and asynchronous messaging (message-oriented) platforms, respectively.

In an MDA design trajectory, a designer should clearly define the abstraction levels at which PIMs and PSMs have to be defined. The choices of platforms should also be made explicit in each step in the MDA design trajectory. Furthermore, the choice of design concepts for platform-independent should be carefully considered, taking into account the common characteristics of the target platforms and the complexity of the transformations that are necessary in order to generate PSMs from PIMs.

4. Application Interaction Systems

Instead of defining the interconnection of application parts directly in terms of the interaction systems provided by a middleware platform, it is possible to identify *application interaction systems* that support application-level interactions between application parts. Figure 5 illustrates the view of an application where an application interaction system is identified.

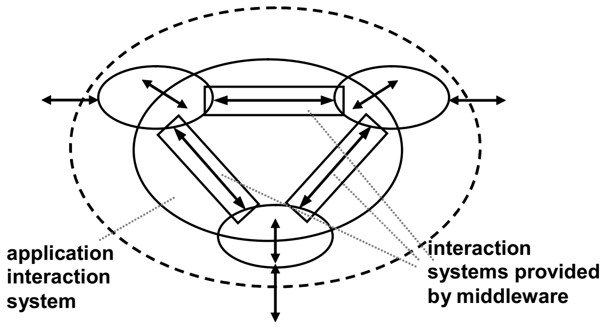
application interaction system
interaction systems provided by middleware

Figure 5. Application interaction systems

Whether or not the design of application interaction systems is considered explicitly depends on the application requirements and on the objectives of the designer [14]. In the following situations, interaction system design should be considered:

- if the relation between system parts is complex. In this case, proper attention should be given to the design of the relation between system parts. This is possible if this relation is made a separate object of design, i.e., if the interaction system of the system parts is considered separately. Consideration of the interaction system is possible at different abstraction levels in order to cope with the complexity of the relation. The interaction system provided by the middleware plays an important role at lower levels of abstraction.
- if it is more likely that interactions are changed than just the contributions to interactions by individual system parts. This is the case if several different middleware platforms are envisioned as alternatives to support the interactions. An interaction mechanism can only be replaced by another equivalent interaction mechanism if the relevant characteristics of the mechanism are clearly indicated in the design. This is naturally supported with interaction system design.

A starting point in the design of an application interaction system is the specification of its service, capturing the succinct description of the interaction system from an external perspective. The design of the application interaction system may, in principle, have any internal structure as long as it provides the required service. For example, it may make use of a data transport service via an application protocol as in a protocol approach [15]. Nevertheless, we observe that the middleware leverages the reuse of a large building block that provides an interoperability architecture across programming languages, operating systems, network technologies and provides facilities to define application-level information attributes. Therefore, we argue that interaction systems provided by the middleware should be considered for building application interaction systems.

Nevertheless, if we structure the design of an application interaction system in terms of the constructs provided by a particular middleware platform, the design of the application interaction system would not be suitable for realizing this design on multiple platforms. Therefore, we define a platform-independent service design in terms of an abstract platform. Later, platform-independent design is realized on top of a concrete-platform.

4.1. Example: Floor-control Service

In order to illustrate the use of an application service in a design trajectory, we introduce our running example, the *floor-control* problem. In this example, several application parts share a set of named resources. Each of these resources can only be used by a single application part at a time, and hence application parts have to coordinate their behaviours in order to ensure that there is no concurrent use of a resource. Application parts are assumed to be cooperative, i.e., they do not use the resources indefinitely. In addition, no pre-emption of control over a resource is necessary.

The service must be specified in such a way that interaction requirements between application parts are satisfied without unnecessarily constraining implementation freedom. This freedom includes the structure of the application interaction system (the system that eventually supports the floor-control service) and other technology aspects such as middleware platforms, operating systems and programming languages. Therefore, services are described in terms of the relations between interactions that occur at the interfaces between the interaction system and the environment. An interaction is an abstract concept that is defined as a common unit of activity performed by two or more system parts. Interactions related in a service description are local, i.e., they occur at a local interface that interconnects an application part and the application interaction system directly [4, 12].

The *floor-control service* relates the following interactions: **request**, **granted** and **free**. These interactions occur at the interfaces between the floor-control service and each of the application parts, which we call *subscribers*. A result of the occurrence of each of these interactions is the establishment of the resource identification and the identification of the subscriber. The latter is implied by the location where the interaction occurs. The following relations between interactions are informally identified:

- Local constraint 1: the occurrence of **granted** follows the occurrence of **request** (for a given resource identification);
- Local constraint 2: the occurrence of **free** follows the occurrence of **granted** (for a given resource identification);
- Remote constraint: a resource is only granted to one subscriber at a time.

The floor-control service is illustrated in Figure 6.

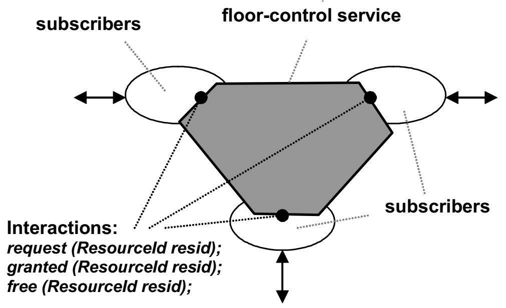

Figure 6. The floor-control service

5. Design Trajectory

5.1. Milestones in Model-driven Design

We define the following milestones in the MDA trajectory:

Service definition. The service definition sets the boundaries of the application interaction system to be designed. Services are specified at a level of abstraction at which the supporting infrastructure is not considered. In our case, the infrastructure is the middleware platform, and therefore, service specifications are middleware-platform-independent by definition. The service concept defines a platform-independent level that is also "paradigm"-independent (as in [2]), in the sense that a service may be implemented by a broad set of middleware platforms that support different interaction patterns. Service definitions are positioned at the top of the design trajectory identified in Figure 4. Application parts that use the service, and therefore rely on the service definition, may be defined at the same level of platform-independence.

Platform-independent service design. The platform-independent service design consists of the *platform-independent service logic*, which is structured in terms of *service components*, and an *abstract-platform definition*. The choice of abstract platform must consider the portability requirements, since it defines the characteristics of the platform upon which service components may rely. The level of abstraction at which the platform-independent service logic is specified depends on the abstract platform definition. Figure 7 illustrates the design trajectory with the service definition and platform-independent service design milestones.

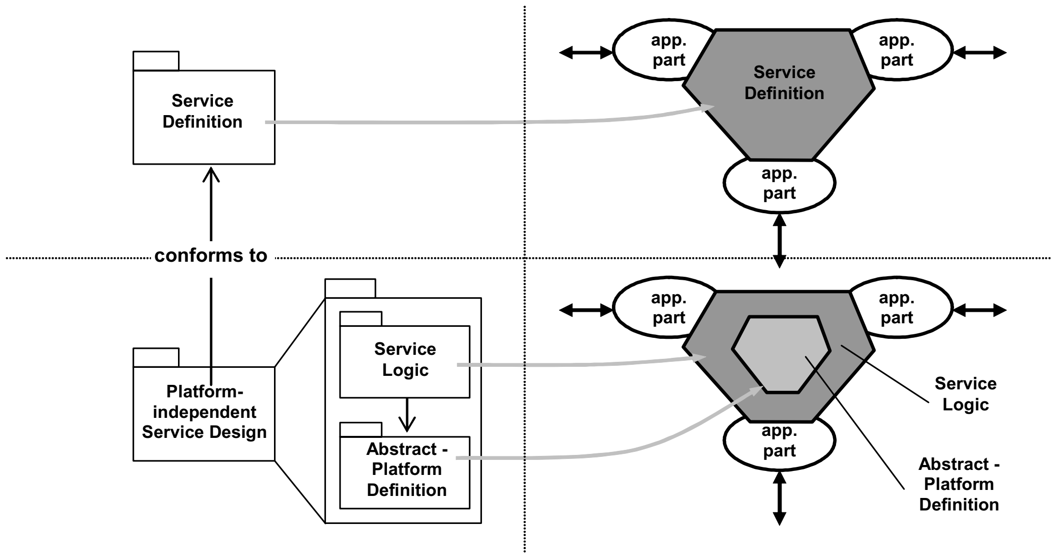

Figure 7. Milestones in the design trajectory

Platform-specific service design. The platform-independent service design is transformed into a platform-specific service design, which is structured in terms of *platform-specific service components* and a *concrete-platform definition*. This transformation may be straightforward when the selected platform corresponds (directly) to the abstract platform definition. This milestone is discussed further in Section 5.4.

The approach outlined in this section suggests a top-down design trajectory, starting from service definition to service design. However, this does not exclude the use of bottom-up knowledge. Bottom-up experience is what allow designers to re-use middleware infrastructures, by defining an abstract platform that can be realized in terms of these concrete middleware platforms, and to find appropriate service designs that implement the required service. Stable abstractions for service design should be derived from knowledge obtained from the solution space (as in a synthesis-based design method [17]).

5.2. Choice of Abstract Platform

The choice of abstract platform defines which (platform-independent) properties or aspects are actually considered and which (platform-dependent) properties or aspects are abstracted from in the design of service components, explicitly defining the notion of platform-independence for the considered design.

In order to define an abstract platform, one must carefully observe:

1. *Portability requirements for the platform-independent design*. The abstract platform should be generic enough to allow a mapping to different target platforms. The actual set of middleware platforms is mostly determined by business and strategic arguments;
2. *The needs of application designers*. The abstract platform should provide facilities that ease platform-independent service design; and,
3. *The extent to which abstract platform and target concrete platforms are different*. It should be possible to obtain platform-specific realizations from platform-independent designs with acceptable quality attributes.

Figure 8 illustrates the factors that influence the choice of abstract platform.

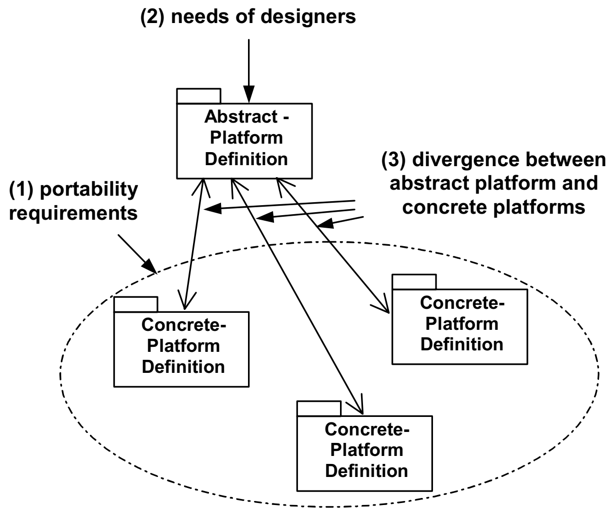
(2) needs of designers
Abstract - Platform Definition
(3) divergence between abstract platform and concrete platforms
(1) portability requirements
Concrete-Platform Definition
Concrete-Platform Definition
Concrete-Platform Definition

Figure 8. Forces in the choice of abstract platform

The forces exercised by factors (2) and (3) are often contradictory:

(i) Raising the provided support to observe the needs of designers may increase the gap between the abstract platform and concrete platforms. This is the case, for example, for the support of multicast message exchange in the abstract platform, when a concrete platform supports only the request/response interaction pattern.

(ii) Reducing the gap between support provided by the abstract platform and concrete platform may lead to an abstract platform that handicaps the designer. This is the case, for example, for a "minimalist" abstract platform that supports a common denominator of a broad class of middleware platforms such as point-to-point one-way message exchange. Patterns such as request/response and multicast message exchange are expected to be built in the service logic.

Differences in the architectural concepts used to build platform-independent designs and those concepts supported by the target platform may result in the use of intricate combinations of implementation constructs in the target platform. This may have an impact on the complexity of the mapping between platform-independent and platform-specific design and on some quality attributes of platform-specific design. It is questionable whether transformations from disparate abstract and concrete platforms would provide platform-specific designs with appropriate quality properties, such as, e.g., traceability from the platform-independent design, time performance, and maintainability.

Shortening the gap between an abstract platform and concrete platforms is a challenging activity. Introducing new concrete platforms because of portability requirements may mean that the gap between the abstract platform and the newly introduced concrete platform is large. Besides that, shortening the gap between an abstract platform and a particular concrete platform may enlarge the gap between the abstract platform and other concrete platforms.

5.3. Abstract Platform Representation

An abstract platform may be defined implicitly by the selection of concepts used to describe platform-independent models. For example, the use of asynchronous message exchange (or "signals") in languages such as SDL [5] or the U2P UML 2.0 submission [18,19] implicitly defines an abstract platform that provides reliable asynchronous message exchange.

An abstract platform may also be explicitly identified in a service definition. When this is the case, it is possible to view platform-independent design as a composition of service components and the abstract platform. We identify the following benefits of defining the service of an abstract platform explicitly:

- Defining an abstract-platform draws attention to considering the trade-offs presented in Section 5.2;
- Abstract-platform service definitions can be used as a starting point for platform-specific realization, as discussed in Section 5.4 and exemplified in Section 6.2; and,
- An abstract-platform service defines explicitly the notion of platform-independence adopted for a design.

5.4. Platform-specific Realization

Platform-specific realization may be straightforward when the selected concrete platform corresponds (directly) to the abstract platform definition.

When this is not the case, more effort has to be invested in platform-specific realization. In general, we distinguish two ways of proceeding with platform-specific realization:

1. A *recursive application of service definition and design*, preserving the border between platform-independent design and the abstract platform. The abstract-platform definition functions as service definition for the recursion. The functionality of the abstract platform is leveraged with the introduction of *abstract-platform service logic*, which is a platform-specific model defined in terms of the concrete platform. (This is equivalent to building-up support in the concrete platform, so that the concrete platform corresponds directly to the abstract platform.)
2. Direct *transformation* with no preservation of the border between abstract platform and platform-independent design. For each concept represented in a platform-independent model, there should be a corresponding concept or a corresponding combination of concepts in the target platform.

Figure 9 illustrates these two approaches to platform-specific realization.

The recursive application of service definition and design (approach 1) provides clear traceability between platform-independent and platform-specific design. Abstract-platform service logic can be reused in the realization of other platform-independent designs that rely on the same abstract platform. An argument against this approach is that it may be harder to satisfy time-performance requirements than with direct transformation (approach 2). Furthermore, recursive application may sacrifice intuitiveness for developers that are accustomed to a particular concrete platform.

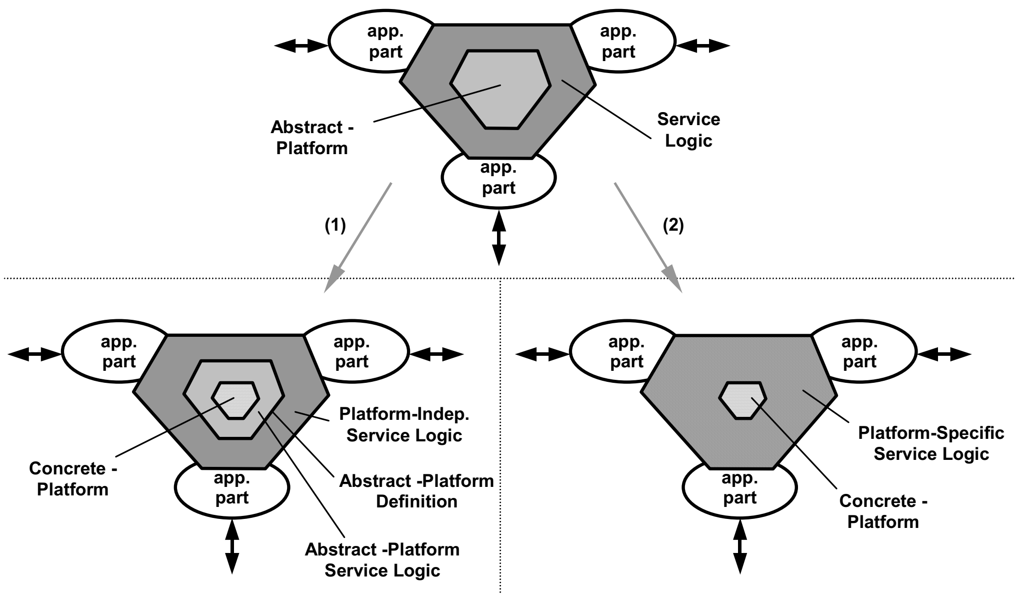

Figure 9. Alternative approaches to platform-specific realization

6. Examples

Using the floor-control service as defined in Section 4.1 as a starting point, we follow the design trajectory for two different abstract platforms: an abstract platform that supports message exchange and an abstract platform that supports the request/response pattern. We consider different design solutions for the floor-control service, illustrating that the service specification is to a large extent implementation-independent. For each platform-independent design obtained, we consider realizations in two concrete platforms: CORBA [10] and the Java Message Service (JMS) point-to-point domain [20]. Figure 10 illustrates the design trajectories followed in our examples.

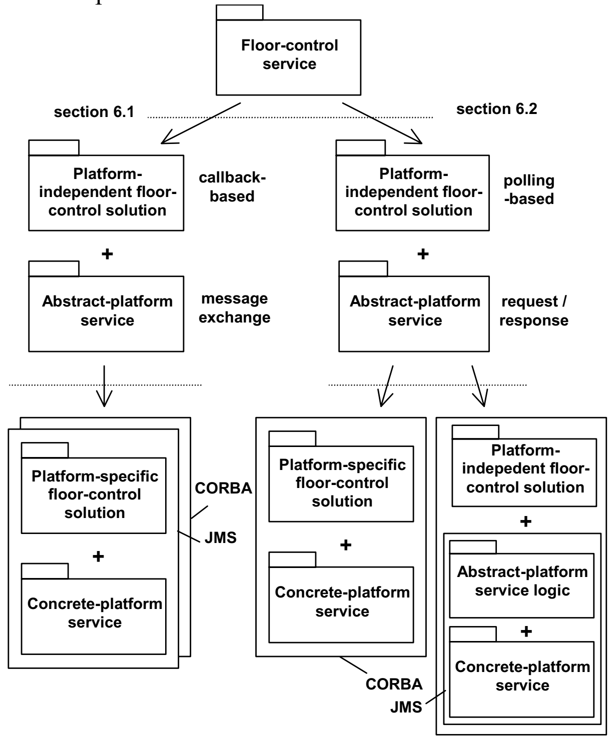
Floor-control service
section 6.1
section 6.2
Platform-independent floor-control solution
callback-based
Platform-independent floor-control solution
polling -based
+
Abstract-platform service
message exchange
Abstract-platform service
request / response
Platform-specific floor-control solution
Concrete-platform service
CORBA
JMS
Platform-specific floor-control solution
Concrete-platform service
CORBA
JMS
Platform-indepedent floor-control solution
Abstract-platform service logic
Concrete-platform service

Figure 10. Example trajectories

6.1. Callback-based solution with Message Exchange Abstract Platform

6.1.1. Abstract Platform: Message Exchange. Initially, let us consider an abstract platform that supports message exchange. We identify two interactions that are related by the abstract platform:

- *send*, with attributes: *destination* and *payload*; and
- *receive*, with attribute *payload.*

An occurrence of *receive* follows an occurrence of *send.* The interaction *receive* is executed at the location specified by the attribute *destination* of *send*. The attribute *payload* represents the information to be sent. The value of the attribute *payload* for an occurrence of *receive* is the value of the attribute *payload* for the related occurrence of *send*.

6.1.2. Platform-independent design. The abstract platform is used in our *callback-based solution* to exchange messages between subscriber service components and the controller service component. The structure of the platform-independent design is depicted in Figure 11.

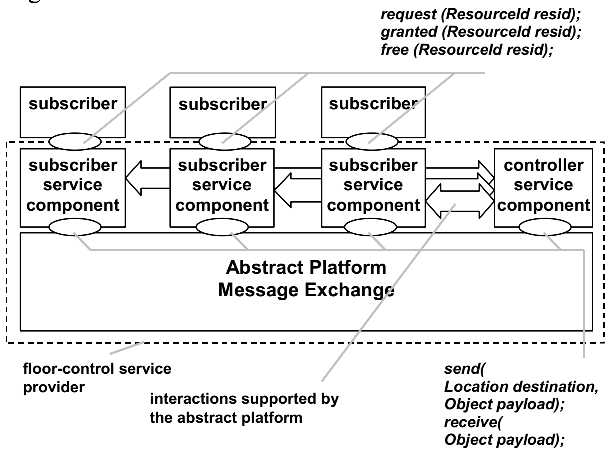
request (ResourceId resid);
granted (ResourceId resid);
free (ResourceId resid);
subscriber
subscriber
subscriber
subscriber service component
subscriber service component
subscriber service component
controller service component
Abstract Platform Message Exchange
floor-control service provider
interactions supported by the abstract platform
send(Location destination, Object payload);
receive(Object payload);

Figure 11. Structure of the callback-based floor-control service provider

The controller service component centralizes the control of the access to the resources. When a subscriber requests for access to a resource, by executing the interaction *request*, the subscriber service component sends a request message to the controller with the identification of the resource. This is done in interaction with the abstract platform through the *send* interaction, which is followed by the occurrence of the *receive* interaction on the interface of the controller service component. Eventually, when the resource is to be granted to the subscriber, the controller sends a *grant message* to the subscriber service component. When the subscriber wants to release the resource, a *free* interaction is executed, resulting in the sending of a *free message* to the controller. A successful execution of a request for a resource is illustrated in Figure 12.

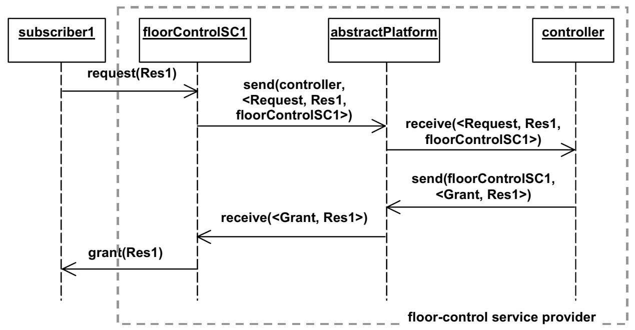
subscriber1
floorControlSC1
abstractPlatform
controller
request(Res1)
send(controller, <Request, Res1, floorControlSC1>)
receive(<Request, Res1, floorControlSC1>)
send(floorControlSC1, <Grant, Res1>)
receive(<Grant, Res1>)
grant(Res1)
floor-control service provider

Figure 12. A resource is requested and granted (Platform-independent design)

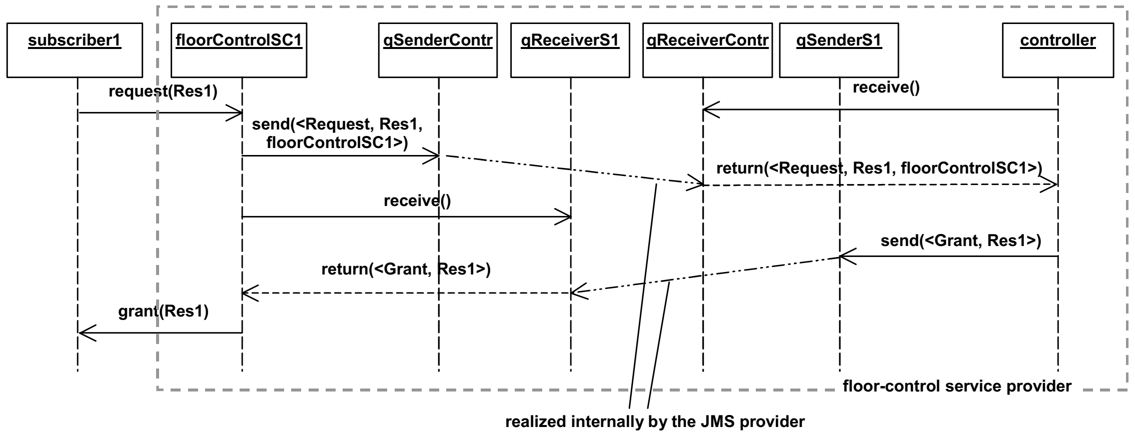

Figure 13. A resource is requested and granted (JMS-specific realization)

6.1.3 Realization. A realization of the platform-independent design in the JMS platform is straightforward. The service provided by JMS corresponds directly to the service provided by the defined abstract platform. A successful execution of a request for a resource in our realization in JMS is illustrated in Figure 13. In the JMS platform, the destination of a message is addressed by a queue identifier. In this solution, there is a queue for messages destined to the controller and a queue for messages destined to each subscriber. The addressing of the destination for a message is done through selection of a queue, and the instantiation of a message producer for the queue (**qSenderContr** for the queue directed to **controller** and **qSenderS1** for the queue directed to **subscriber1**).

The realization in the CORBA platform can be obtained through a simple transformation: message exchange is realized through an operation invocation with no return parameters. A successful execution of a request for a resource in our realization in the CORBA platform is illustrated in Figure 14.

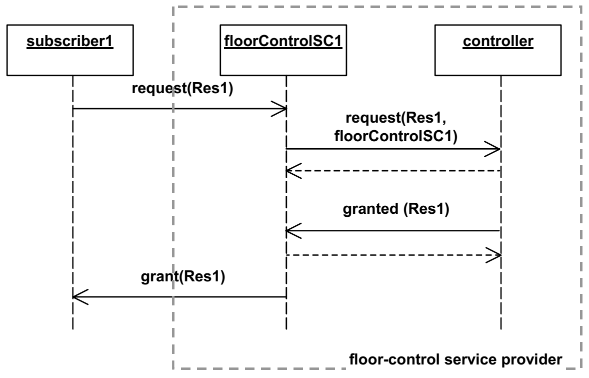

Figure 14. A resource is requested and granted (CORBA-specific realization)

For the CORBA realization, we could have also considered the use of the CORBA Notification Service [11] in a similar way as we have used JMS to accomplish message exchange. This illustrates our observation that there are many possible ways to realize a platform-independent design even for a particular concrete platform.

6.2. Polling-based solution with Request/Response Abstract Platform

6.2.1. Abstract Platform: Request/Response. Let us consider an abstract platform that supports the request/response pattern. We identify four interactions that are related to each other through the abstract platform:

- *request*, with attributes: *target*, *operation* and *argument_list*. The attributes represent, respectively, the identifier of the target object, the identifier of the requested operation and the argument list for the request;
- *request_ind*, with attributes: *operation* and *argument_list*;
- *response*, with attribute *return_parameters*, which represents the list of return parameters; and,
- *response_ind*, with attribute *return_parameters*.

The occurrence of *request_ind* follows the occurrence of *request*, the occurrence of *response* follows the occurrence of *request_ind*, and the occurrence of *response_ind* follows the occurrence of *response*.

This is a generalization of the service provided by request/response platforms. These platforms provide some infrastructure to generate customized stubs that in conjunction with the middleware core provide specializations of the service as presented in this section.

6.2.2. Platform-independent design. The abstract platform is used in our *polling-based solution* to enable the subscriber service components to issue invocations to the controller. The structure of the platform-independent design is depicted in Figure 15, which is identical to

Figure 11 except for the abstract platform and its primitive interactions.

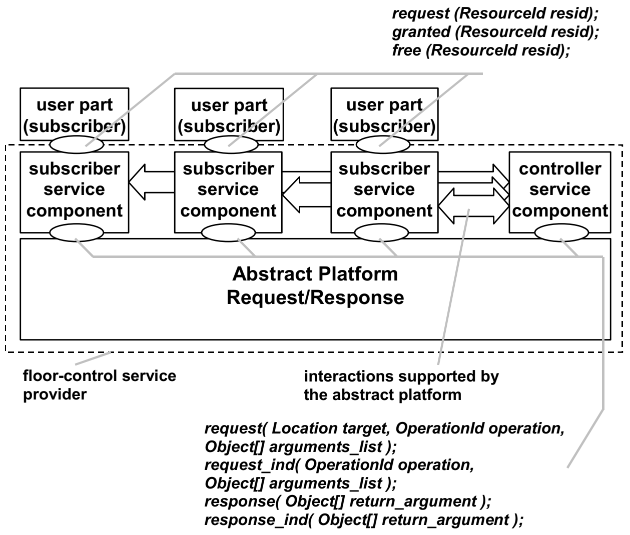
request (ResourceId resid);
granted (ResourceId resid);
free (ResourceId resid);
user part (subscriber)
user part (subscriber)
user part (subscriber)
subscriber service component
subscriber service component
subscriber service component
controller service component
Abstract Platform Request/Response
floor-control service provider
interactions supported by the abstract platform
request(Location target, OperationId operation, Object[] arguments_list);
request_ind(OperationId operation, Object[] arguments_list);
response(Object[] return_argument);
response_ind(Object[] return_argument);

Figure 15. Structure of the callback-based floor-control service provider

The subscriber service components poll the controller for a certain resource by invoking its operation *request_permission*, which returns the Boolean value *true* when the resource is available, and *false* otherwise. When the subscriber wants to release the resource, the operation *free* of the controller's interface is invoked. A successful execution of a request for a resource is illustrated at the top of Figure 16.

6.2.3. Realization. A realization of the platform-independent design in terms of the CORBA platform is straightforward. The realization in terms of the JMS platform deserves more attention, since this platform does not support the request/response pattern directly.

We have applied the approach 1 to realization as presented in Section 5.4: the abstract platform service specification is used as a starting point for a recursive application of service design. The diagram at the bottom of Figure 16 illustrates a successful execution of a request for a resource, in a realization with the abstract platform realized in terms of the JMS platform. The occurrence of a *request* interaction results in the sending of a *request* message to the controller, containing the identification of the request, the name of the operation to be invoked, and the parameters for the operation. The identification of the

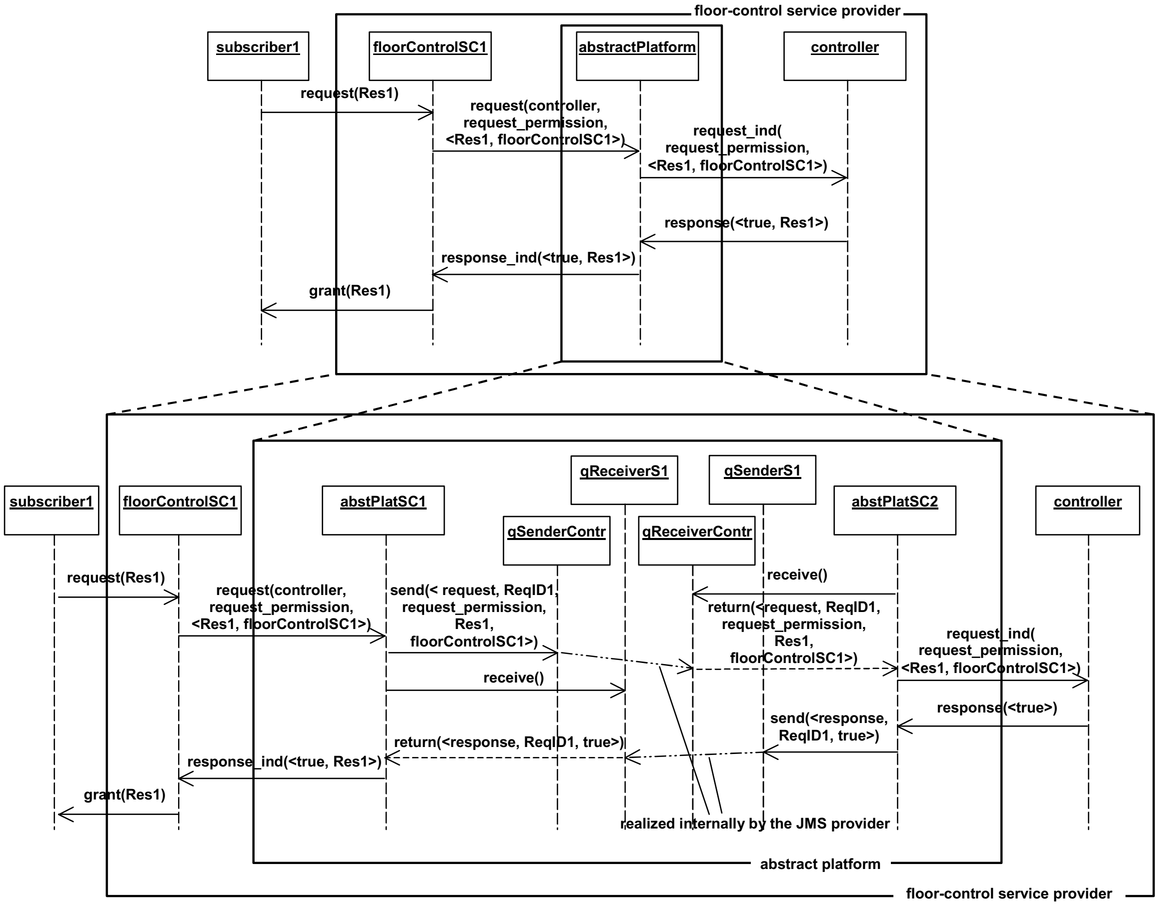
floor-control service provider
subscriber1
floorControlSC1
abstractPlatform
controller
request(Res1)
request(controller, request_permission, <Res1, floorControlSC1>)
request_ind(request_permission, <Res1, floorControlSC1>)
response(<true, Res1>)
response_ind(<true, Res1>)
grant(Res1)
subscriber1
floorControlSC1
abstPlatSC1
qSenderContr
qReceiverS1
qReceiverContr
qSenderS1
abstPlatSC2
controller
request(Res1)
request(controller, request_permission, <Res1, floorControlSC1>)
send(< request, ReqID1, request_permission, Res1, floorControlSC1>)
receive()
receive()
return(<request, ReqID1, request_permission, Res1, floorControlSC1>)
request_ind(request_permission, <Res1, floorControlSC1>)
response(<true>)
send(<response, ReqID1, true>)
return(<response, ReqID1, true>)
response_ind(<true, Res1>)
grant(Res1)
realized internally by the JMS provider
abstract platform
floor-control service provider

Figure 16. A resource is requested and granted

request is used by the abstract platform service components to correlate request and response messages.

A solution based on direct transformation (approach 2) would also be possible, embedding functionality to correlate request and response in the floor control service components. In this case, the structure of the platform-independent design would not be directly recognizable in the platform-specific design.

6.3. Symmetric solutions

Both platform-independent solutions we have explored are asymmetric implementations of the floor-control service. Asymmetric solutions are characterized by separate controller and subscriber roles. The controller centralizes the coordination of access to shared resources, while subscribers must request the controller for access to a resource.

In addition to the asymmetric solutions we have presented, we identify a class of *symmetric* solutions to the floor-control service. In symmetric solutions, there is no controller, and all application parts have identical roles in the coordination. An example of a symmetric solution is based on token passing. In this solution, a list with the set of available resources circulates among the subscribers. Each subscriber examines the list with the set of identifiers of available resources, removes the identifier of the resource desired and forwards the list by invoking an operation on the interface of the following subscriber. When a subscriber wants to release a resource, it inserts the identifier of the resource to be released in the list.

These solutions have been investigated and are approached in the same way as the asymmetric solutions presented here. They are further ignored in this paper.

6.4. Discussion

Among the solutions discussed for the floor-control problem, the floor-control service is a stable abstraction, and shields the design of subscribers from the particular way in which the service is implemented. We have shown that the floor-control service is neutral, both with respect to *premature commitments to particular design solutions* (callback-, polling-, or token-based) and with respect to *premature commitments to a particular middleware interaction pattern* (as provided by CORBA and JMS).

It is irrelevant for the design of subscriber application parts whether the design of the floor-control solution is symmetric or asymmetric, callback-, polling-, or token-based, or whether the platform is CORBA or JMS.

For the design of the application interaction system itself, we have used abstract platform definitions. This allowed us to target CORBA and JMS from the same platform-independent design. Moreover, by using the abstract platform service specification as a starting point for a recursive application of service design (approach 1 for platform-specific realization), we have obtained software components that can be reused on top of different platforms.

Our approach focuses on the behavioural aspects of platform-independent design. Therefore, in the presentation of our examples, we have not explored issues related to the treatment of information value types. Nevertheless, we acknowledge that these issues are an important aspect of MDA development.

7. Conclusions

We have argued the case for a more prominent role of service specifications and interaction system design in the model-driven development of distributed applications. The service concept allows us to provide support for a designer to define precisely the notion of platform-independence adopted for a design, based on the definition of an abstract platform.

By defining application interaction systems with service specifications, and designing application parts that rely on the service definition, we achieve a high level of platform-independence. Consequently, the design of application parts can be reused across a large set of middleware platforms. Furthermore, particular implementations of the application interaction system are irrelevant for the design of application parts that use the interaction system.

We have identified the use of the service concept in different milestones of the model-driven development trajectory. Service specifications have to be expressed in a suitable modelling language. Cariou et al. [3] have recently explored the notion of “medium” which corresponds to the notion of *application interaction system* we adopt, focussing on the use of UML to represent such media. We intend to propose extensions or usages of UML with respect to the representation of the service concept, both for the representation of the service of application interaction systems and the service of abstract and concrete platforms.

Acknowledgements

This work is partly supported by the European Commission, in context of the IST project MODA-TEL (http://www.modatel.org) and the ‘Telematica Instituut’, in the context of the Dutch national project ArCo (http://arco.ctit.utwente.nl/). We also thank the anonymous reviewers for their useful comments and suggestions for improvements.

References

[1] R. J. Allen, and David Garlan, "A Formal Basis for Architectural Connection", *ACM Transactions on Software Engineering and Methodology*, vol. 6, n. 3, July 1997, pp. 213-219.

[2] C. Burt et al., "Quality of Service Issues Related to Transforming Platform Independent Models to Platform Specific Models", *Proceedings Sixth International Conference on Enterprise Distributed Object Computing*, Lausanne, Switzerland, Sept. 2002, pp. 212-223.

[3] E. Cariou, A. Beugnard, and J. M. Jézéquel, "An Architecture and a Process for Implementing Distributed Collaborations", *Proceedings Sixth International Conference on Enterprise Distributed Object Computing*, September 2002, Lausanne, Switzerland, Sept. 2002, 132-143.

[4] L. Ferreira Pires, *Architectural Notes: a framework for distributed systems development*, Ph.D. Thesis, University of Twente, Enschede, The Netherlands, 1994, available at http://www.cs.utwente.nl/~pires/thesis/

[5] International Telecommunications Union (ITU), *Specification and description language (SDL)*, ITU-T Recommendation Z.100, Geneva, Switzerland, Aug. 2002.

[6] Microsoft Corporation, *Microsoft .NET Remoting: A Technical Overview*, July 2001, available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/hawkremoting.asp

[7] Object Management Group, *Model driven architecture (MDA)*, OMG document ormsc/01-07-01, July 2001.

[8] Object Management Group, *Generic RFP template*, OMG document ab/02-04-06, April 2002.

[9] Object Management Group, *CORBA Component Model*, v3.0, OMG document formal/02-06-65, July 2002.

[10] Object Management Group, *Common Object Request Broker Architecture: Core Specification*, Version 3.0, OMG document formal/02-12-06, Dec. 2002.

[11] Object Management Group, *Notification Service Specification*, v1.0.1, OMG document formal/02-08-04, Aug. 2002.

[12] D.A.C. Quartel. *Action relations. Basic design concepts for behaviour modelling and refinement,* Ph.D. Thesis, University of Twente, Enschede, The Netherlands, 1998, at http://www.cs.utwente.nl/~quartel/publications/PhD/

[13] D.A.C. Quartel, L. Ferreira Pires, M. van Sinderen, H.M. Franken, and C.A. Vissers. On the role of basic design concepts in behaviour structuring. *Computer Networks and ISDN Systems*, vol. 29 (1997) 413-436.

[14] M. van Sinderen. *On the Design of Application Protocols*. Ph.D. Thesis. University of Twente, Enschede, The Netherlands, March, 1995, available at http://www.cs.utwente.nl/~sinderen/publications/thesis.html.

[15] M. van Sinderen and L. Ferreira Pires, "Protocols versus objects: can models for telecommunications and distributed processing coexist?", *Proceedings Sixth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, Tunisia, October 1997, 8-13.

[16] M. van Sinderen, L. Ferreira Pires, C.A. Vissers, and J.-P. Katoen, "A design model for open distributed processing systems". *Computer Networks and ISDN Systems* 27 (1995) 1263-1285.

[17] B. Tekinerdogan, *Synthesis-Based Software Architecture Design*, Ph.D. Thesis, University of Twente, March, 2000, available at http://www.cs.utwente.nl/~bedir/PhDThesis/

[18] U2 Partners. *UML 2.0: Infrastructure, 3rd revised submission*, OMG document ad/2003-03-01, March, 2003.

[19] U2 Partners' *UML 2.0: Superstructure, 3rd revised submission*, OMG document ad/2003-04-01, April, 2003.

[20] Sun Microsystems, *Java(TM) Message Service Specification Final Release 1.1*, April 2002, available at http://java.sun.com/products/jms/docs.html

[21] C. A. Vissers, L. Ferreira Pires, D. A. Quartel, M. van Sinderen. *The Architectural Design of Distributed Systems*, Lecture Notes, University of Twente, Enschede, The Netherlands, Nov. 2002.

[22] C.A. Vissers, and L. Logrippo, "The importance of the service concept in the design of data communications protocols". *Proceedings Fifth IFIP WG6.1 International Conference on Protocol Specification, Testing and Verification*, June 1985, 3-17.

[23] C. A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma, "Specification styles in distributed systems design and verification". *Theoretical Computer Science*, 89:179–206, 1991.

[24] World Wide Web Consortium, *Web Services Description Language (WSDL) 1.1*, W3C Note, March 2001, available at http://www.w3.org/TR/wsdl

[25] World Wide Web Consortium, *Simple Object Access Protocol (SOAP) 1.1*, W3C Note, May 2000, available at http://www.w3.org/TR/SOAP/