Abstract Interactions and Interaction Refinement in Model-Driven Design

João Paulo Almeida, Remco Dijkman, Luís Ferreira Pires, Dick Quartel, Marten van Sinderen
Centre for Telematics and Information Technology, University of Twente, The Netherlands
{j.p.andradealmeida, r.m.dijkman, l.ferreirapires, d.a.c.quartel, m.j.vansinderen}@utwente.nl

Abstract

In a model-driven design process the interaction between application parts can be described at various levels of platform-independence. At the lowest level of platform-independence, interaction is realized by interaction mechanisms provided by specific middleware platforms. At higher levels of platform-independence, interaction must be described in such a way that it can be further refined and realized onto a number of different middleware platforms, each with its particular interaction mechanisms and implementation constraints. In this paper we investigate concepts that support interaction design at various levels of middleware-platform-independence. Also, we propose design operations for interaction refinement. The application of these operations to source designs results in target designs that take into account implementation constraints imposed by platforms, while preserving characteristics prescribed in source designs.

1. Introduction

In our previous work [1], we have argued that the design of a system can be considered at various levels of platform-independence in a model-driven design process. An initial design in a model-driven design process is given at a high level of platform-independence, meaning that it considers little or none of the constraints that a platform imposes on the way in which that design can be implemented. Examples of such platform constraints are prescriptions of mechanisms that must be used to realize interactions between system parts in a design (e.g. operation invocation, message passing or publish-subscribe queues). Other examples are constraints on the threading models that can be used to realize concurrent execution of behaviours (e.g. single-threaded, thread per request or thread pool). During the design process, a designer must gradually consider these constraints, and the means to incorporate them into designs. Eventually, this should lead to a design at a sufficiently low level of platform-independence such that the realization of the design becomes straightforward.

For these reasons, a model-driven design process requires design concepts and supporting modelling languages that are abstract enough to construct designs in which no specific platform constraints are imposed. At the same time, such concepts should be expressive enough to allow the construction of designs at a sufficiently detailed level to describe how the design can be realized.

The first goal of this paper is to identify and motivate concepts that support interaction design at various levels of platform independence. In order to abstract from particular interaction mechanisms at a high level of platform-independence, we consider that application parts interact through *abstract interactions*. Designers relate abstract interactions to their realizations in middleware platforms by applying design operations.

The second goal of this paper is to introduce design operations that can be used to transform a source design at a certain level of platform-independence into a target design at a lower level of platform-independence. These design operations preserve the characteristics prescribed by a source design and gradually incorporate platform constraints into target designs. We focus on constraints and concepts that address the communication aspects of middleware platforms.

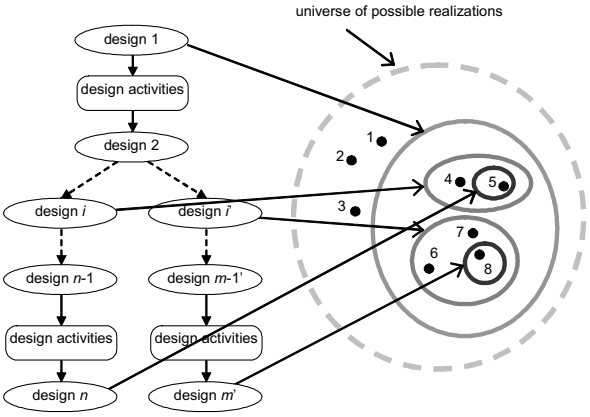
The remainder of this paper is structured as follows: Section 2 characterizes the model-driven design process. Section 3 presents an instance of the design process that we use as example throughout the paper. This example consists of alternative transformations for the same platform-independent design. Section 4 proposes candidate design concepts. Section 5 proposes design operations, using these to transform designs in our example. Section 6 revisits the example, exploring the transformations not worked out in section 5. This serves to show the variety of platform constraints that can be accommodated in the design process. Section 7 discusses some limitations of our approach and compares the proposed design concepts with those underlying UML and SDL. Section 8 discusses related work. Finally, section 9 provides our conclusions and identifies some future work.

Proceedings of the 2005 Ninth IEEE International EDOC Enterprise Computing Conference (EDOC'05)
0-7695-2441-9/05 $20.00 © 2005 IEEE

2. Model-driven design process

We characterize a model-driven design process as a series of design steps, each of which results in a design of the system. Designs are represented in a symbolic artefact called a *model*. For each design step, *design activities* are executed, which consist of transformation and assessment activities [19]. A *transformation activity* is a generic design activity that entails the production of a target design on basis of a source design and requirements. An *assessment activity* is a generic design activity that comprises the evaluation of the target design as outcome of the transformation activity.

During the design process, transformation activities incorporate a number of design decisions to a design, which add characteristics that will eventually be assigned to the realization of a design. Different design decisions lead to different alternative realizations. The reduction of the realization space imposed by successive design decisions is depicted in Figure 1 (inspired by [19]).



1..8 – examples of elements from the universe of possible realizations
4..8 – examples of elements from the set of possible realizations defined by design 1
4..5 – examples of elements from the set of possible realizations defined by design *i*
6..8 – examples of elements from the set of possible realizations defined by design *i'*
5 – example of element from the set of possible realizations defined by design *n*
8 – example of element from the set of possible realizations defined by design *m'*

Figure 1. Reduction of realization space for designs at different levels of abstraction

Design decisions taken in a design step should meet two requirements for the design process to make progress [12]: (i) they must contribute to satisfying requirements that have not yet been fulfilled, and (ii) they must preserve the characteristics present in the source design, i.e., the target design should conform to the source design. The latter requirement reveals the importance of *conformance assessment* in a design step. This is reflected in our approach in the use of design operations that result in conformant refinements of designs (see section 5).

Design decisions should eventually lead to a design that defines all relevant characteristics of an acceptable realization of the system. The platform on which the design will be realized partly determines which design decisions can be made. Similarly, design decisions determine possible platforms on which the design can be realized.

For the purpose of this paper, we assume that distributed applications are ultimately realized in some object- or component-middleware platform that supports basic interconnection between distributed application parts, such as CORBA/CCM [16], .NET (Remoting) [13], and Web Services [25, 26]. We call the middleware platform on which the design will be implemented the *realization platform* (or *platform* for short).

A platform provides reusable constructs for an application designer, who does not have to be concerned about the implementation of these constructs. For example, a designer of CORBA objects does not have to be concerned about the GIOP protocol and the marshalling and demarshalling of invocations. By providing particular realization constructs, a realization platform imposes a number of constraints on designs. These constraints may apply to the (types of) entities that can be used in a design, the way they interact with each other, their life-cycle, structure, behaviour, etc. The constraints imposed by the realization platform must be incorporated in designs (through design steps). This leads to (platform-specific) designs that can be implemented in the realization platform with relatively little effort. These designs are such that each concept in the design either corresponds to a construct that is provided by the realization platform, or is part of a pattern of concepts that corresponds to a construct that is provided by the realization platform.

Designs at a high-level of abstraction that can be realized onto different platforms are called *platform-independent designs*. The corresponding models are called *platform-independent models* (PIMs) in the MDA [14]. The level of platform-independence of a design depends on the sets of design concepts, combinations of concepts or patterns used, which constitute what we call an *abstract platform*. An abstract platform is an abstraction of infrastructure characteristics assumed for models of an application at a certain level of platform-independence [1]. For example, if a platform-independent design contains application parts that interact through operation invocations (e.g. in a UML [15] model), then operation invocation is a characteristic of the abstract platform. Capabilities of a realization platform are used during platform-specific realization to support this characteristic of the abstract platform. For example, if CORBA [16] is selected as a target platform, this characteristic can be mapped onto CORBA operation invocations.

3. Running example: the design of a conferencing application

Our running example consists of the design of a conferencing application. This application facilitates the interaction of users residing in different hosts. Let us suppose that, initially, the designer describes the application as a composition of conference participants, a conference manager and a conference service provider. The service provider is described solely from its external perspective, revealing its interfaces and relating interactions that occur at these interfaces. At this point in the design process, the characteristics of the internal design of the conference service provider are not revealed. In addition, we assume that the interfaces are described in terms of abstract interactions and interaction relations, which do not prescribe any particular interaction mechanism. The abstract platform at this level of abstraction supports the interactions between application parts and the conference service provider. Figure 2 shows how a snapshot of this design (D_0) could be visualized. It distinguishes three conference participants and one conference manager.

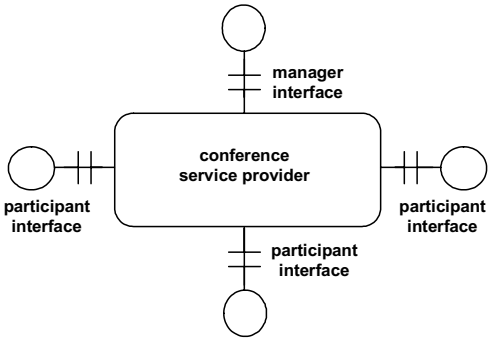


Figure 2. A snapshot of design D_0

We distinguish two basic approaches to further refine design D_0:

(i) *interaction refinement* [9], in which case a designer refines the interactions between the application parts and their environment without changing the granularity of the parts, i.e., without decomposing the parts into smaller parts, or;

(ii) *entity refinement* (which is called *interaction allocation and flowdown* in [24]), in which case the designer decomposes the application parts into smaller parts and allocates the existing interactions to these parts. In this case, the interactions remain unchanged, except for the introduction of new (internal) interactions between the smaller parts.

Figure 3 depicts these approaches schematically. It also shows that interaction refinement and entity refinement can be applied in combination.

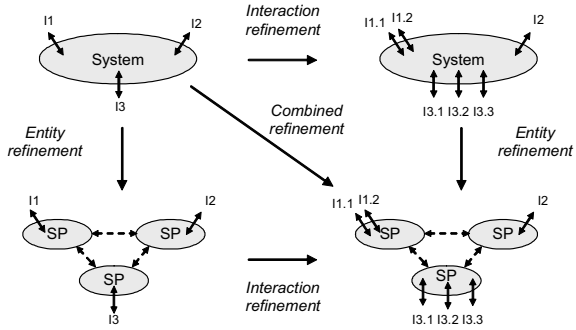


Figure 3. Approaches to system refinement [9]

We consider several alternative transformations of design D_0, according to the *interaction refinement* approach. The following alternatives show how different platform characteristics influence the refinement process:

1. We refine D_0 into a design D_1 that uses an abstract platform that supports *operation invocation between objects* and supports *multiple operation interfaces per object*. The conference service provider is not decomposed, and is directly implemented as a single object in the realization.
2. We refine D_1 into a design D_2, and as in design step (1) described above, we use an abstract platform that supports operation invocation. In this case, however, we add the platform-imposed constraint that *the abstract platform supports only a single operation interface per object*.
3. We refine D_0 into a design D_3, and as in design step (1) described above, we use an abstract platform that supports operation invocation between objects. The abstract platform supports a single operation interface per object. In this case, however, we add a platform-imposed constraint that *participants and managers are located in so-called 'thin clients', which cannot be used as targets for operation invocation.*
4. We refine D_0 into a design D_4 that uses an abstract platform that supports *asynchronous messaging* between objects. *The abstract platform supports multiple messaging queues*. The conference service provider is not further decomposed.

The abstract platform used in design D_2 facilitates the realization of this design in a CORBA platform (which offers only a single operation interface per CORBA object). The abstract platform used in design D_3 facilitates the realization of this design in a Web Services platform, e.g. with the conference service provider hosted in a J2EE platform, with 'thin clients' running in Mobile Information Device Profile (MIDP) devices [21]. The abstract platform used in D_4 facilitates the realization of this design using the Java Message Service (JMS) [20] or the CORBA Event Service.

Figure 4 depicts these alternative transformations steps and the resulting designs.

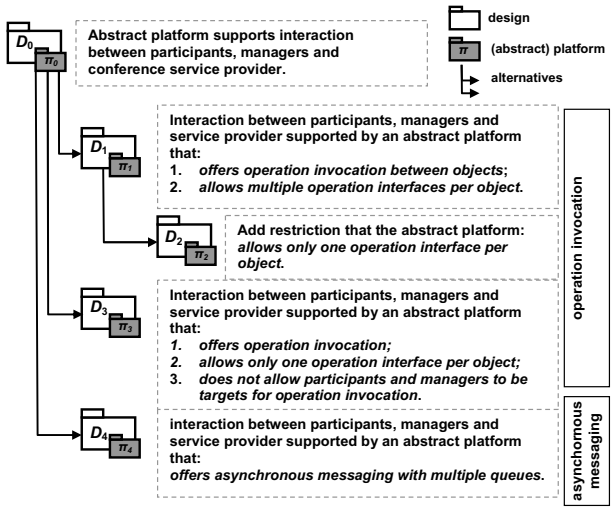


Figure 4. Alternative design steps

By applying interaction refinement, the alternative transformations presented in this section consider the use of different abstract platforms for distributing the interactions between participants, managers and the conference service provider. An alternative is to follow the *entity refinement* approach, decomposing the conference service provider and using an abstract platform to support the distribution of the various internal components of the conference service provider. This allows many alternative realizations of the initial design, onto different abstract platforms. For example, the conference service provider may be further decomposed into a centralized or distributed, symmetric or asymmetric design, and different abstract platforms may be used to support the interactions of the objects that implement it [2]. We acknowledge that the entity refinement approach is useful in the model-driven design process, but we do not discuss this type of refinement further in this paper, since this has been the subject of our previous work (in [2]). Therefore, we concentrate on the role of interaction refinement in the model-driven design process.

4. Concepts for abstract platform design

In this section we generalize the alternative design steps of section 3 to derive requirements for concepts for design at various levels of platform-independence. Also, we propose some basic design concepts that fulfil the requirements. We assume that design concepts should cover both the behaviour and structural aspects of systems.

4.1. Requirements

We claim that the example from section 3 motivates requirements for design concepts that are not considered in current state of the art modelling languages.

Requirements for interactions. The example motivates the need for an interaction concept that abstracts from a particular interaction mechanism, because at the highest level of platform-independence no interaction mechanism should be chosen. The example presents an operation invocation and an asynchronous messaging mechanism for the eventual implementation of the design. However, an abstract interaction concept should abstract from these interaction mechanisms and allow the designer to use any mechanism for the implementation of the design. Therefore, we propose an interaction concept that only represents:

- the identity of the interaction;
- the successful occurrence of the interaction;
- the information that is available to the interacting parties as a result of the interaction and the location at which this information is available; and
- optionally the direction in which the information flows.

Such a concept abstracts from:

- roles that the interacting parties play in the interaction (e.g. initiator or responder);
- aspects of interaction mechanisms that we have yet to decide upon (e.g. whether an interaction corresponds to an operation invocation or a message being passed, whether queues are used to temporarily store messages, or whether an operation is blocking or non-blocking).

Requirements for interfaces. The example also motivates the need for abstract interfaces that abstract from a particular interaction mechanism through which communication takes place. An abstract interface abstracts from any constraints that an interaction mechanism may impose on the way in which that interface can be used. An example of such a constraint is that, at an interface, only remote procedure calls can be responded to, while no remote procedure calls can be invoked. A CORBA interface is an example of a mechanism that imposes these constraints. We propose an abstract interface concept that only represents:

- the identity of the interface;
- the interactions that are supported by the interface, as well as the relations between these interactions; and
- the party that interacts via the interface.

Such a concept abstracts from:

- any constraints on the interaction mechanisms that are available at the interface (e.g. only remote procedure calls can occur at this interface);
- any constraints on the role that the owner of the interface may play in interactions that occur at that

interface (e.g. the entity that owns the interface can only play the role of responder in interactions that occur at this interface);
- the addressing scheme that is used to identify the interface (e.g. whether the interface is identified by a URI or a CORBA object reference).

4.2. Basic design concepts

We claim that the basic design concepts from Figure 5 satisfy the requirements identified in section 4.1, because they define abstract interaction and interface concepts. The concepts from this figure are an adapted version of the RM-ODP basic modelling concepts [11] as explained in [7].

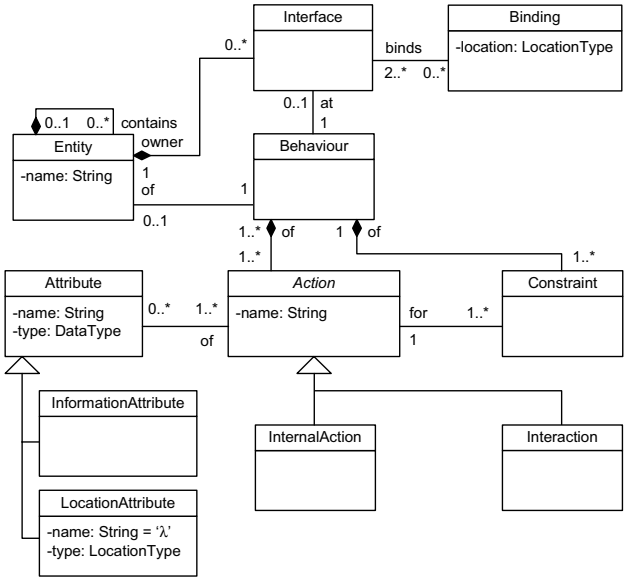


Figure 5. Conceptual Model

An *entity* is a logical or physical carrier of behaviour. It is uniquely identified by a name. Entities can contain other entities to represent how they are composed. Entities also contain *interfaces* that represent parts of the mechanisms that they use to interact with other entities. Interfaces can be connected by a *binding*, which represents a shared mechanism for interaction. The parts of this shared mechanism correspond to the interfaces that the binding connects. Note that a binding does not represent something *in between* the interfaces. The bound interfaces themselves constitute the mechanism.

Entities have a behaviour in the context of which they perform actions. Interfaces also have behaviours, which are abstractions of the behaviour of an entity. These abstractions represent the actions that entities perform in the context of a binding. We call an action that is performed by a single entity an *internal action*. We call an action that is performed by multiple entities in collaboration an *interaction*.

If an interaction occurs, its results are available to all its participants. If an interaction does not occur, no result is established. Hence, none of the participants can refer to any (intermediate) result. The possible results of an interaction are represented by information attributes. If an interaction occurs, the values of its information attributes represent the result of the interaction. An interaction can also be associated with a location attribute that represents the possible locations at which it can occur. If an interaction occurs, the value of its location attribute represents the location at which its results are available. This location identifies a binding.

Constraints on actions determine when these actions are allowed to occur (*causality conditions*) and what kinds of results are possible as the outcome of an action (*attribute constraints*). Each behaviour that participates in an interaction can define its own constraint on the occurrence of that interaction. We call that constraint an *interaction contribution*.

Each interacting entity constrains the attributes established as result of an interaction: a party may offer a set of values, accept a set of values, or both. These constraints on values supply different ways of cooperation [18], namely, *value passing*, *value checking* and *value generation*. Value passing occurs when an interacting party offers a value and the other parties accept this value. Value checking occurs when all interacting parties offer the same value. In value generation, the interacting parties offer a range of acceptable values and the interaction happens if it is possible to establish a value that matches all requirements.

4.3. Application of design concepts to D_0

Figure 6 presents a snapshot of the structural aspects of D_0 in terms of the basic concepts described above. An entity is represented by a rectangle with cut-off corners that contains entity's name. An interface is represented as a line that is connected to the owner of the interface by another line. A binding is represented by a dashed line that connects the bound interfaces. Bindings are annotated with their location.

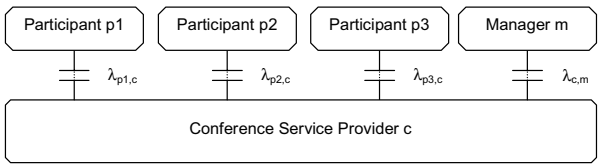


Figure 6. D_0 Snapshot

We identify the following (value passing) interactions:

- sendmsg interactions, which occur at the bindings between participants and the conference service provider ($\lambda_{pn,c}$ in Figure 6). These interactions result in the establishment of a message to be sent (the information attribute i_{msg}). In this interaction, information flows from participants to the conference service provider;
- receivemsg interactions, which occur at the bindings between participants and the conference service provider ($\lambda_{pn,c}$). These interactions result in the establishment of the message received. In the receivemsg interaction, information flows from the conference service provider to a participant;
- the include interaction, which occurs at the binding between the manager and the conference service provider ($\lambda_{c,m}$). This interaction establishes the identification of a participant (the information attribute $i_{particip}$) that is to be included in the conference. In this interaction, information flows from the manager to the conference service provider;
- the exclude interaction, which occurs at the binding between the manager and the conference service provider ($\lambda_{c,m}$). This interaction establishes the identification of a participant (the information attribute $i_{particip}$) that is to be excluded from the conference. In this interaction, information flows from the manager to the conference service provider.

The following causality conditions apply to the interactions:

- the occurrence of receivemsg interactions follows the occurrence of a sendmsg interaction; receivemsg interactions occur at the bindings between participants currently included in the conference and the conference service provider;
- the occurrence of include eventually leads to a participant being included in the conference, and;
- the occurrence of exclude eventually leads to a participant being excluded from the conference.

Figure 7 represents graphically part of the interactions and constraints of D_0 graphically. For simplicity, it only shows two participants and only the interactions necessary for one participant *p1* to send a message to the conference. Also, it only shows the include interaction with the conference manager. For the sake of conciseness the figure only represents one instance of occurrence of these interactions, i.e., it ignores that more instances of these interactions may occur.

A behaviour is represented by a rounded rectangle that carries the name of its corresponding entity or interface. An internal action is represented by a circle drawn inside the behaviour in the context of which it is defined. An interaction contribution is represented by a semi-circle drawn on the border of the behaviour in the context of which it is defined. An interaction is represented as dashed lines that connect the interaction contributions that form the interaction. Attributes are drawn inside a box, along with the name of the action to which they belong. Attributes are attached to an action by a dashed line. Action constraints are drawn inside the box that is attached to the action (for attribute constraints), or they are represented by an arrow that means that the action can only occur after the action at the origin of the arrow has occurred (for causality conditions). A constraint of an interaction is drawn inside the behaviour that is responsible for enforcing that constraint. For example, sendmsg enables receivemsg and it is the conference service provider's responsibility to ensure that this constraint is enforced. Also, it is the conference service provider's responsibility to ensure that the receivemsg interaction only occurs with participants that are in the set of conference participants (participantSet). In this paper we do not present the precise way to represent constraints (we refer to [18] for more information about this aspect of design).

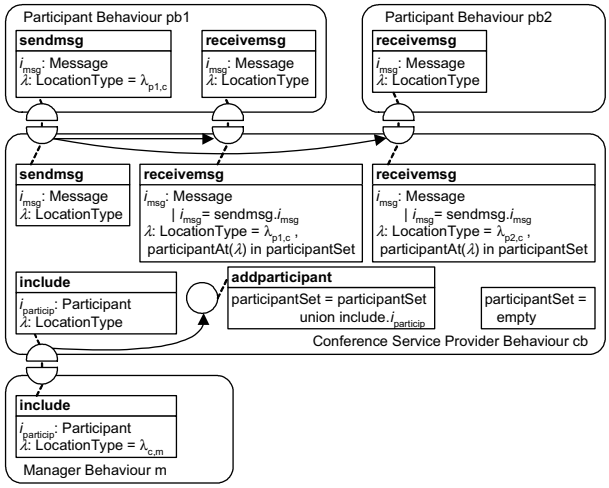


Figure 7. Conference System Behaviour

5. Design operations

A design that does not correspond directly to a realization in a selected target platform can be further transformed using the following design operations: (inter)action refinement, binding and interface decomposition, binding and interface merging, and entity merging. We present each of these operations in the following sub-sections, by motivating and illustrating them with the conference application and using the concepts presented in section 4.2.

5.1. Action refinement

If an action (i.e., either an interaction of internal action) can not be supported by a construct from the

realization platform, we must refine that action into multiple actions that *can* be directly supported by the realization platform.

An action can not be refined into an arbitrary set of actions and constraints, because the refined behaviour must preserve the characteristics that the original behaviour prescribed. [17] explains how designs, constructed with an extension of the concepts from section 4.2, can be refined correctly. Basically, each action is refined into a group of *final actions* that correspond to the completion of that action and *inserted actions* that do not. Since the final actions correspond to the original action, they must together enforce the same constraints and deliver the same results as the original action.

Table 1 presents the rule for refining an action into multiple actions, making certain design decisions.

Table 1. Action refinement: definition

Input	Any action *a.*
Design decisions	Any (as long as constraints imposed by conformance relation are respected, see below).
Output	A group of actions that capture design decisions made. This group of actions consists of *final actions* that correspond to the completion of the original action *a* and *inserted actions* that do not. Final actions must together enforce the same constraints and deliver the same results as the original action *a* [17].

5.2. Action refinement example

In our conference example, none of the realization platforms support the abstract interaction concept directly through the supported interaction mechanisms. All the mechanisms in the considered platforms require additional design decisions, such as, defining the party responsible for initiating interaction. Therefore, the behaviour of a platform's interaction mechanisms is often defined at a level of abstraction at which multiple lower level actions are executed by the interacting parties. For example, asynchronous messaging mechanisms identify an interaction for a party to send a message and an interaction for a party to receive a message. A remote procedure invocation mechanism identifies an interaction for a client to issue a request, an interaction for a server to receive a request, an interaction for a service to respond to a request and an interaction for a client to receive the response to the request

Table 2 illustrates how action refinement can be applied to refine an interaction into multiple interactions that form a remote invocation.

Table 2. Action refinement: transformation

Input	Any interaction *i* in which a value is passed from one party to another.
Design decisions	Operation invocation is used to realize interaction. The entity that passes value in the interaction initiates communication.
Output	The interaction *i* is refined into: a invocation_req interaction, a invocation_ind interaction, a invocation_rsp interaction and a invocation_cnf interaction. invocation_ind is a final interaction, all others are inserted interactions.

5.3. Binding and interface decomposition

The consideration of platform characteristics in a design may require bindings and interfaces to be decomposed into multiple bindings and interfaces. This operation must be applied to a binding and its interfaces in a source design, if the interaction mechanisms that a realization platform provides can not directly support the binding.

Table 3 presents the rule for binding and interface decomposition. The entities and bindings by which a binding is replaced in the refined design must connect the entities that correspond to the original entities of the abstract design. Otherwise, the refinement does not preserve the connectivity of the original design.

Table 3. Binding decomposition: definition

Input	Any binding λ (and interfaces associated with it).
Design decisions	Any (as long as constraints imposed by conformance relation are respected, see below).
Output	Entities that are connected through the original binding λ are connected through a configuration of bindings and entities that replace λ.
Implications for behaviour domain	Interactions that occur at binding λ should occur at locations introduced by bindings or entities that replace λ.

Binding decomposition and action refinement are often coupled, because, if a binding is refined, interactions that occurred at that binding must be refined into actions that can be assigned to the refinement of that binding.

5.4. Binding decomposition example

We obtain design D_1 from D_0 in two steps. Table 4 shows the transformation used in the first step, in which the bindings from D_0 are decomposed into multiple entities.

Table 4. Binding decomposition: transformation

Input	Any binding λ (and interfaces associated with it) between two entities e_1 and e_2.
Design decisions	Operation invocation is used.
Output	An entity e_π that supports operation invocation is introduced. This entity is connected to e_1 through a binding $\lambda_{\pi 1}$ and connected to e_2 through a $\lambda_{\pi 2}$.
Implications for behaviour	(Inter)actions that replace original interactions that occur at binding λ should occur at $\lambda_{\pi 1}$ or $\lambda_{\pi 2}$ or e_π.

Figure 8 illustrates this decomposition step graphically.

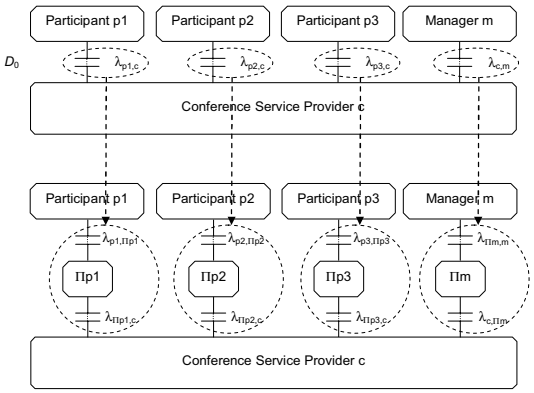


Figure 8. Action refinement and binding decomposition applied to D_0

The interactions that occurred at the original binding are refined according to the rule from Table 2. The sendmsg interactions which occur at bindings $\lambda_{pn,c}$ are refined into:

- a invocation_req interaction, which occurs at binding $\lambda_{pn,\Pi pn}$ between a participant and an entity that is part of the abstract platform (see Figure 8). This interaction results in the establishment of the name of an operation to be invoked, arguments for the invocation, and an identifier for the invocation i_{id}. This identifier is unique in the context of the binding and is used to distinguish between multiple simultaneous invocations[1]. In this refinement, the name of the operation is sendmsg (not to be confused with the sendmsg interaction from Figure 7) and the argument is the value of information attribute i_{arg}. In our case this argument will carry a more concrete representation of the message that is sent.
- a invocation_ind interaction, which follows the occurrence of invocation_req. The invocation_ind interaction occurs at binding $\lambda_{\Pi pn,c}$ between an entity that is part of the abstract platform and the conference service provider (see Figure 8). The results of this interaction are the same as the results of the invocation_req interaction;
- a invocation_rsp interaction, which occurs at the same binding at which the invocation_ind interaction occurs. Since the sendmsg interaction only consists of an information flow from a participant to the conference service provider, the response does not have to carry any information;
- a invocation_cnf interaction, which occurs at the same binding at which the invocation_req interaction occurs. This interaction follows the occurrence of the invocation_rsp interaction.

The include and exclude interactions are refined in a similar way. The receivemsg operation differs in that it is targeted at participants. Because of space restrictions we omit the discussion of this refinement.

Figure 9 represents part of the refined behaviour. However, it only shows one participant. The figure illustrates that the abstract platform behaviour can accept invocation_req interactions at both the binding with the participant and the binding with the conference service provider, because it does not restrict the location λ at which the interaction can take place. Note that this means that the invocation_req interaction contribution is a part of two interactions, one with the service provider and one with the participant. Upon engaging in a invocation_req, it causes an invocation_ind at the other binding. The figure also illustrates that, after engaging in a invocation_ind interaction in which the sendmsg operation is referenced, the conference service provider enables an invocation_req, in which the receivemsg operation is referenced.

In Figure 9, invocation_ind with a value of sendmsg for i_{op} is a final action for sendmsg from Figure 7. Similarly, invocation_ind with a value of receivemsg for i_{op} is a final action for receivemsg from Figure 7. Now we can verify that, after abstracting from inserted actions invocation_req, invocation_rsp and invocation_cnf, the final actions enforce the same constraints as the actions for which they are final actions. For example, the constraint from Figure 7 that receivemsg is caused by sendmsg is also enforced by the final actions for receivemsg and sendmsg from Figure 9.

[1] This identifier is either implicit or explicit in realization platforms. For example, a CORBA client using the Dynamic Invocation Interface (DII) manipulates the identifier of a request explicitly. In contrast, for a client using compiled stubs the identifier of a request is implicit and corresponds to the thread in which the local stub method is invoked.

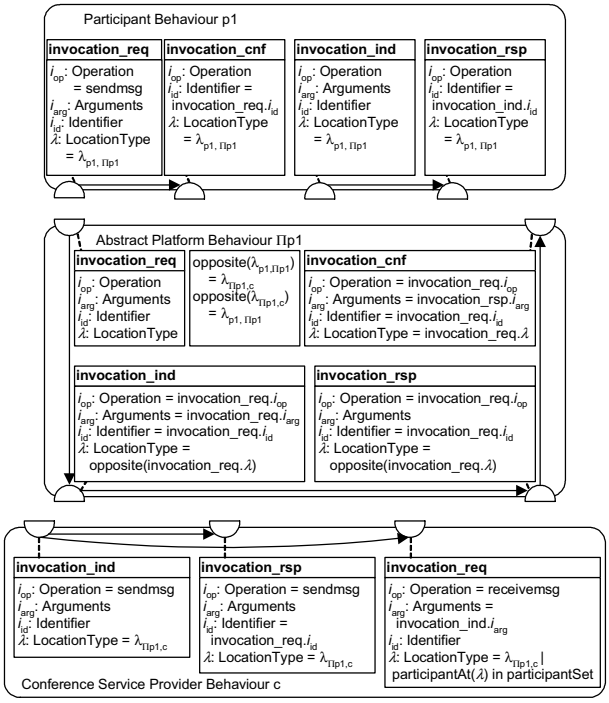


Figure 9. Refined behaviour

In the design depicted in Figure 8 and Figure 9 the targets of operation invocation are implied by bindings in which an invocation_req occur. For example, if a invocation_req occurs at binding $\lambda_{p1,\Pi p1}$, the invocation is targeted at the conference service provider. We can further transform this design by generalizing the behaviour of the entities that make up the abstract platform so that they support operation invocations between two arbitrary entities. This results in a better matching between this behaviour and the behaviour of realization platforms (such as, CORBA, Web Services, Java RMI). This generalization is accomplished by adding an information attribute (i_{dst}) to invocation_req, which identifies the binding at which a corresponding invocation_ind should occur. This attribute is defined by the entity that initiates an invocation. Figure 10 illustrates this.

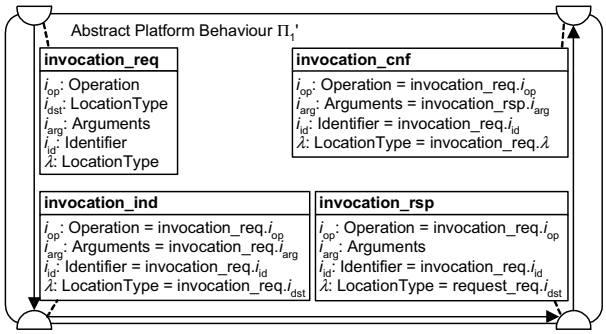


Figure 10. Invocation target as attribute i_{dst}

5.5. Entity merging

The consideration of platform characteristics to a design may require entities to be merged into a single entity. This operation must be applied, if a realization platform supports multiple entities in a design as a single entity. Table 5 presents the rule for entity merging. The resulting entity has all the bindings that the original entities had. Similarly, the resulting entity carries all the behaviours of the original entities.

Table 5. Entity merging: definition

Input	Any set of entities e_i.
Design decisions	None.
Output	A merged entity e replaces the original entities e_i.
Implications for behaviour domain	Merged entity carries behaviour of entities e_i.

5.6. Entity merging example

Figure 11 shows the application of entity merging in our example. Entities Π_{p1}, Π_{p2}, Π_{p3} and Π_{p4} are merged into an entity Π_1'. Entity merging does not affect the behaviour domain. The behaviour of the original entities is carried by the merged entity.

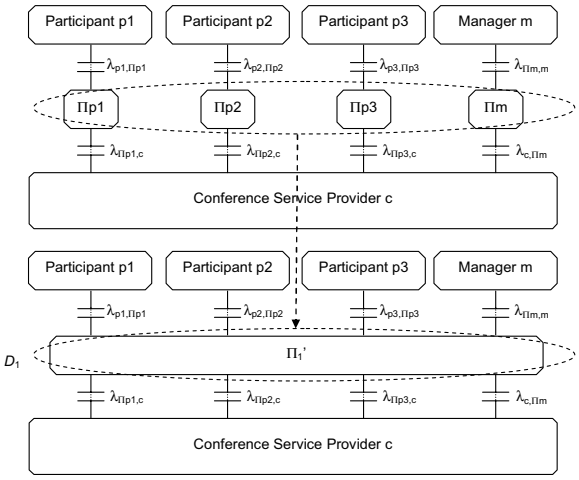


Figure 11. Entity merging to obtain D_1

5.7. Binding and interface merging

The consideration of platform characteristics to a design may require interfaces to be merged into a single interface. This operation must be applied to some interfaces and their bindings, if a realization platform imposes constraints on the number of interfaces that can be attached to an entity and the design violates these constraints. Merging of interfaces may require the

interactions that occur at these interfaces to be refined, because interactions with the same name could originally be distinguished by the interface names. However, if the interfaces are merged, they can not be distinguished anymore. Table 6 presents the rule for binding and interface merging.

Table 6. Binding and interface merging: definition

Input	Any set of bindings λ_i between the same set of entities.
Design decisions	None.
Output	A binding λ replaces the bindings λ_i.
Implications for behaviour domain	Behaviour preserves distinction between interactions. For example, information attributes can be used to distinguish interactions that occur at different original bindings λ_i.

5.8. Binding and interface merging example

We use binding and interface merging to obtain D_2 from D_1. In platform Π_2, an entity is not allowed to have more than one interface through which it plays the responding role in invocations. Therefore, multiple interfaces through which an entity plays a responding role must be merged into a single interface (the corresponding bindings are also merged). This step is depicted in Figure 12.

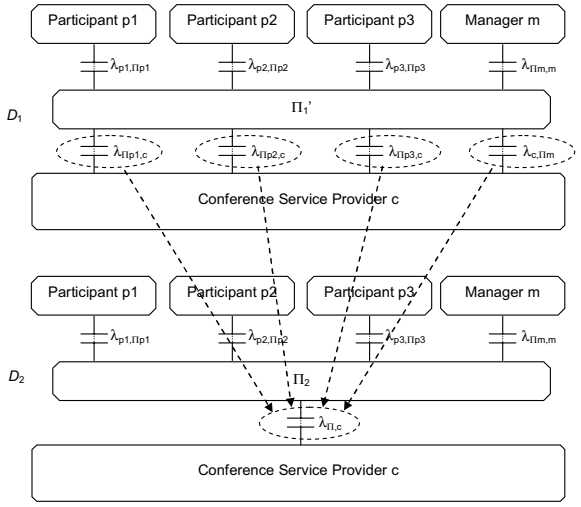


Figure 12. Binding and interface merging applied to D_1, resulting in D_2

The application of the binding merging operation consists of replacing bindings $\lambda_{\Pi p1,c}$, $\lambda_{\Pi p2,c}$, $\lambda_{\Pi p3,c}$, and $\lambda_{c,\Pi m}$ by $\lambda_{\Pi,c}$ and should be reflected in the behaviour of entity Π_1' by replacing the bindings being merged by $\lambda_{\Pi,c}$. In addition, invocation_req interactions that occur at bindings $\lambda_{\Pi p1,c}$, $\lambda_{\Pi p2,c}$, $\lambda_{\Pi p3,c}$, and $\lambda_{c,\Pi m}$ (in D_1) are replaced by interactions at binding $\lambda_{\Pi,c}$ which have an additional information attribute i_{dst} that can have the values $\lambda_{p1,\Pi p1}$, $\lambda_{p2,\Pi p2}$, $\lambda_{p3,\Pi p3}$, and $\lambda_{\Pi m,m}$. respectively. This ensures that the interactions can still be distinguished as belonging to different original bindings. For example, an invocation_req interaction that originally occurred at binding $\lambda_{\Pi p1,c}$ is replaced by an invocation_req interaction that occurs at binding $\lambda_{\Pi,c}$ and has the value $\lambda_{p1,\Pi p1}$ for i_{dst}. We say that in this way the topology of the original structure is preserved.

5.9. Realization of abstract platforms

By applying the design operations we have presented, a designer gradually refines a design into a design whose implementation onto a realization platform is straightforward. For example, the implementation of platform D_2 on a CORBA platform is straightforward, because we can apply the following transformation: each abstract platform entity from D_2 is implemented as a remote procedure invocation mechanism that is supported by CORBA; each interface is implemented as a CORBA operation interface on the client or on the server side, as it is specified in IDL; and each interaction is implemented as an interaction in the remote procedure invocation mechanism (invocation request, indication, response or confirmation). Figure 13 illustrates a realization of the design from Figure 12 and the corresponding behaviour on a CORBA platform. In the figure, the location of a binding corresponds to an entry in the CORBA naming service.

```
Participant
  …
  Object c_object = resolve("lambda_Pi2,c");
  ConferenceServiceProvider c =
    ConferenceServiceProviderHelper.narrow(c_object);
  c.sendmsg(i_arg);
  …
```

```
ConferenceServiceProvider
  …
  void sendmsg(Arguments i_arg){
    …
  }
  …
```

Figure 13. Example of Abstract Platform Realization

6. The example revisited

In section 5, we have discussed how the design operations can be applied to obtain designs D_1 and D_2. In this section, we show how designs D_3 and D_4 can be obtained from the same platform-independent design D_0.

For D_3, we use an abstract platform that supports operation invocations between objects to realize the interactions between participants, managers and the conference service provider. In this design *participants and managers are located in so-called 'thin clients', which cannot be used as targets for operation invocation*.

The refinement of interactions sendmsg, include and exclude is identical to the refinement we have presented

earlier for D_2. The refinement of receivemsg differs significantly, since this interaction is realized through a polling scheme. The receivemsg interaction is refined into the following interactions:

- a invocation_req interaction, which occurs at binding $\lambda_{pn,\Pi pn}$ between a participant and an entity that represents the abstract platform. This interaction results in the establishment of the name of an operation to be invoked, in this case receivemsg_poll, and an identifier for the invocation, with the same role as the identifier used in section 5.4;
- a invocation_ind interaction, which follows the occurrence of invocation_req. The invocation_ind interaction occurs at binding $\lambda_{\Pi pn,c}$ between an entity that represents the abstract platform and the conference service provider;
- A invocation_resp interaction, which occurs at the same binding at which the invocation_ind interaction occurs. The information attribute consists of a Boolean value ($i_{\text{isavailable}}$), which indicates whether a message is available, and the message (i_{arg}), if available;
- A invocation_cnf interaction, which occurs at the same binding at which the invocation_req interaction occur. This interaction follows the occurrence of the invocation_resp interaction.

A recursion in the refined behaviour is necessary, when the value of the $i_{\text{isavailable}}$ information attribute of invocation_cnf is false. The final action that corresponds to the original interaction is invocation_cnf with $i_{\text{isavailable}}$ equals true. Similarly to the case of design D_2, we can further transform this design by generalizing the behaviour of the entities representing the abstract platform so that they support operation invocations between two arbitrary entities.

For D_4, we use an abstract platform that supports *asynchronous messaging* between objects. *The abstract platform supports multiple messaging queues*. The sendmsg interaction is refined into the following interactions:

- a data_req interaction, which occurs at binding $\lambda_{pn,\Pi pn}$ between a participant and an entity that represents the abstract platform. This interaction results in the establishment of the message to be sent;
- a data_ind interaction, which follows the occurrence of data_req. The data_ind interaction occurs at binding $\lambda_{\Pi pn,c}$ between an entity that represents the abstract platform and the conference service provider.

Similar refinements apply to the other interactions, with the exception of receivemsg, in which case the data_req is directed from the conference service provider to the abstract platform and the data_ind is directed from the abstract platform to a conference participant. Each pair of participant and service provider shares a message queue.

The data_ind interaction is the final interaction in the refinements. Depending on the constraints on the original interaction, it may be necessary to insert additional interactions to preserve the constraints in the source design. For example, if a participant performs an action that follows the occurrence of the sendmsg interaction, it is necessary to insert interactions in the target design to inform the participant that data_ind has occurred. This can actually be seen in the refinement framework as a refinement of the causality relation between sendmsg and the actions that depend on its occurrence [17].

We summarize the operations we have shown in this paper in Figure 14:

- the transformation marked by ❶ consists of interaction refinement (with a request/response pattern), generalization and entity merging;
- the transformation marked by ❷ consists of binding merging;
- the transformation marked by ❸ consists of interaction refinement (with a polling scheme), and generalization;
- the transformation marked by ❹ consists of interaction refinement (asynchronous messaging).

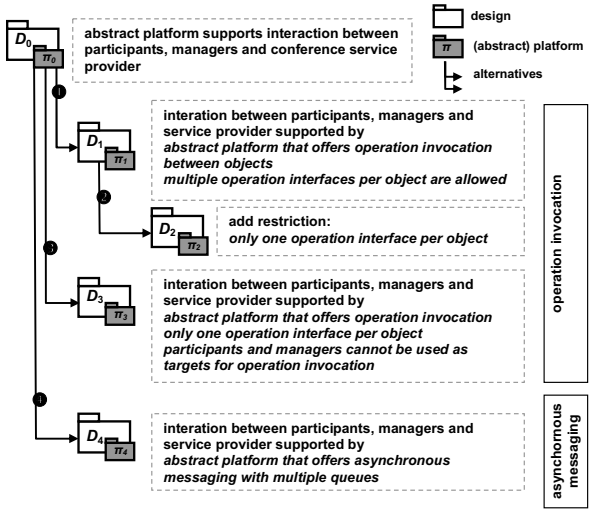


Figure 14. Design operations and the designs

7. Discussion

In this section, we discuss some issues in the use of the design concepts proposed in section 4 and compare the abstract interaction concept we adopt with the interaction concepts underlying UML [15] and SDL [10].

7.1. Modelling failure

In our approach, an interaction represents the successful completion of a shared activity. When the activity being modelled fails to complete, we say that the

abstract interaction does not occur. If it is necessary to represent the failure of an activity explicitly, the failure should be modelled as an interaction, which can only occur if the interaction that models the successful completion of the activity does not occur.

A consequence of this modelling choice is that failure is perceived by all interacting entities. Therefore, it is not possible to model partial failures of a shared activity in this way. If it is necessary to model partial failure explicitly, the designer must model the shared activity at a lower level of abstraction, e.g., by modelling an entity between interacting entities and describing partial failure through the behaviour of this entity.

7.2. Value generation

As discussed in section 4.2, the notion of interaction we adopt can be used to model value generation. Value generation can be used to describe complex shared activities at a high-level of abstraction. For example, it is possible to model the negotiation of quality-of-service contracts between parties with their own requirements using a single interaction. However, value generation should not be used indiscriminately, since it may require sophisticated mechanisms for its reliable realization when distribution must be considered.

7.3. Concepts derived from operation invocation and message passing

Popular modelling languages, such as UML [15] and SDL [10], use basic interaction concepts that are derived from operation invocation and message passing mechanisms.

Operation invocation and message passing concepts represent both the direction in which information flows and the initiating and responding roles for an interaction. Therefore, these force a designer to prescribe the direction of an interaction and roles in an interaction at all levels of platform-independence. This, for example, forces a designer to decide at a high level of platform-independence, whether information is obtained by an entity using a callback or a polling mechanism. For both mechanisms, information flows in the same direction, but in one the sender of the information takes the initiative, while in the other the recipient takes initiative. We claim that such a decision often depends on characteristics of the realization platform, which a designer should not be forced to consider at a high level of platform-independence. For example, a designer may choose between a callback and a polling mechanism for performance reasons. If CORBA is used as a realization platform, using a callback mechanism requires the server-side part of an ORB to be installed on the side of the recipient of the information. This may be problematic, e.g. for mobile devices with few resources. Installing the server-side part of an ORB is not required when the designer chooses for a polling mechanism.

In addition, languages that use operation invocation and message passing concepts often define some details of the mechanisms that realize operation invocation and message passing. For example, in SDL, interacting parties exchange messages through queues of infinite length. Messages exchanged are always delivered unaltered and in sequence. These assumptions may not match the characteristics of a target realization platform, forcing a designer to bridge a large gap between the design and its realization. This significantly decreases the benefit of a model-driven design approach.

Other languages, like UML, leave such aspects for the designer to decide (with semantic variation points). UML defines that "*The means by which requests are transported to their target depend on the type of requesting action, the target, the properties of the communication medium, and numerous other factors. In some cases, this is instantaneous and completely reliable while in others it may involve transmission delays of variable duration, loss of requests, reordering, or duplication.*" [15] Such variation points must be decided upon by the application designer (or tool designer), even at a high-level of platform-independence. This is because different decisions for these aspects would result in different application models. We can conclude that semantic variation points allow designers to select between alternative semantics for some of its constructs, but does not allow designers to abstract from the alternatives, e.g., at a high-level of platform-independence (ambiguity is different from abstraction).

8. Related work

Design transformations in which implementation constraints are incorporated have been proposed earlier, for example, in the LOTOSphere [4] project. Some of the design operations we have presented here have been inspired by the transformations described in [19]. These transformations have been developed to bridge the abstraction gap between formal languages and implementation environments, which is in some aspects similar to the gaps between platform-independent models and platform-specific models that have to be bridged by transformations in MDA. The difference between the transformations in [19] and the design operations proposed here is that the former transformations do not consider middleware technologies as implementation environments (platforms) and therefore they cannot be directly applied to our situation.

Similarly to our approach, the authors of [3] propose a framework in which design concerns can be introduced at subsequent levels of models, which they call *strata*. Our

work contributes to theirs, in that we provide more details about the design concepts that could be used at the different strata and the conformance relations that can exist between them.

We approach interaction refinement from the perspective of architectural design. Several authors approach interaction refinement from a pure formal perspective (e.g., [5], [6]). We believe that, in many cases, these approaches make simplifications at the cost of the usefulness of the formal model for pragmatic engineering purposes (as argued in [23]).

9. Conclusions

Most efforts related to transformations in model-driven design and MDA focus on the languages, methods and tools for the specification of model transformation. These efforts are complementary to the work presented in this paper, since the design operations we have defined can be used to derive model transformation specifications that could be implemented by tools.

This paper contributes to the understanding of the design operations that are applied by transformation in a model-driven design approach. Furthermore, we argue that suitable notions of conformance between source and target designs are necessary if we want to reach a mature model-driven design process. This paper gives some ideas on how these notions of conformance can be defined and enforced.

We have shown that the interaction concept and interaction refinement design operations can be used to realize a platform-independent design in multiple realization platforms. This is possible because interaction can be modelled at a high level of abstraction with the design concepts proposed here. This level of abstraction is higher than the level of abstraction that can be obtained with concepts that correspond closely to operation invocation and asynchronous messaging mechanisms, such as those underlying UML and SDL. This implies that proper language support for these abstract concepts has to be provided. In this paper we have applied the notation of the Interaction Systems Design Language (ISDL) [22] to represent these abstract concepts and have shown that this notation copes with our modelling requirements.

The design concepts we have described in this paper represent the behaviour of the system given a certain system configuration of entities, interfaces and bindings, i.e., ignoring the actions necessary to modify the system structure during execution. In [8], we described design concepts that can be used to describe some of these actions, like the dynamic creation and destruction of entities, interfaces and bindings. The application of the interaction refinement operations presented in this paper when considering these dynamic modifications in the system configuration still remains to be investigated.

Furthermore, we intend to develop tool support that implements the design operations presented in this paper in terms of (semi-)automated model transformations.

Acknowledgements

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

References

[1] J.P.A. Almeida, R. Dijkman, M. van Sinderen and L. Ferreira Pires, "On the Notion of Abstract Platform in MDA Development," *Proc. 8th IEEE Int'l Conf. on Enterprise Distributed Object Computing (EDOC 2004)*, IEEE CS Press, Sept. 2004, pp. 253-263.

[2] J.P.A. Almeida, M. van Sinderen, L. Ferreira Pires and D. Quartel, "A systematic approach to platform-independent design based on the service concept," *Proc. 7th IEEE Int'l Conf. on Enterprise Distributed Object Computing (EDOC 2003)*, IEEE CS Press, Sept. 2003, pp. 112-134.

[3] C. Atkinson, T. Kuhne, "Aspect-Oriented Development with Stratified Frameworks", *IEEE Software*, 20(1), IEEE CS Press, 2003, pp. 81-89.

[4] T. Bolognesi, J. van de Lagemaat, and C. Vissers, eds., *LOTOSphere: Software Development with LOTOS*, Kluwer Academic Publishers, 1995.

[5] E. Brinksma, B. Jonsson, and F. Orava, "Refining Interfaces of Communicating Systems", In Proc. of the Int'l Joint Conf. on Theory and Practice of Software Development (TAPSOFT'91): Lecture Notes in Computer Science 494, Springer-Verlag, 1991, pp. 297-312.

[6] M. Broy, "(Inter-)action refinement: The easy way", *Program Design Calculi*, Springer NATO ASI Series, Series F : Computer and System Sciences, 118, Springer-Verlag, New York, 1993, pp. 121-158.

[7] R.M. Dijkman, D. Quartel, L. Ferreira Pires, M. van Sinderen, "A Rigorous Approach to Relate the RM-ODP Enterprise and Computational Viewpoint," *Proc. 8th IEEE Int'l Conf. on Enterprise Distributed Object Computing (EDOC 2004)*, IEEE CS Press, Sept. 2004, pp. 187-200.

[8] R.M. Dijkman, "A Basic Design Model for Service-Oriented Design," *ArCo Project Deliverable ArCo/WP1/T1/D2/V1.00*, University of Twente, Enschede, The Netherlands, November 2003.

[9] C.R.G. de Farias, *Architectural design of groupware systems: a component-based approach*, Ph.D. thesis, University of Twente, Enschede, The Netherlands, May 2002.

[10] ITU-T, *Recommendation Z.100 – CCITT Specification and Description Language*, International Telecommunications Union (ITU), 2002.

[11] ITU-T / ISO, *Open Distributed Processing - Reference Model – All Parts*, ITU-T X.901-4 | ISO/IEC 10746-1 to 10746-4, Nov. 1995.

[12] H. Kremer, *Protocol Implementation: Bridging the gap between Architecture and Realization*, Ph.D. thesis, University of Twente, Enschede, The Netherlands, Oct. 1995.

[13] Microsoft Corporation, *Microsoft .NET Remoting: A Technical Overview*, July 2001; http://msdn.microsoft.com/library/en-us/dndotnet/html/hawkremoting.asp
[14] Object Management Group, *MDA-Guide, V1.0.1*, omg/03-06-01, June 2003.
[15] Object Management Group, *UML 2.0 Superstructure*, ptc/03-08-02, Aug. 2003.
[16] Object Management Group, *Common Object Request Broker Architecture: Core Specification, Version 3.0*, formal/02-12-06, Dec. 2002.
[17] D. Quartel, L. Ferreira Pires and M. van Sinderen, "On Architectural Support for Behaviour Refinement in Distributed Systems Design," *Journal of Integrated Design and Process Science*, 6 (1), Society for Design and Process Science, 2002.
[18] D. Quartel, L. Ferreira Pires, M. van Sinderen, H. Franken and C. Vissers, "On the role of basic design concepts in behaviour structuring," *Computer Networks and ISDN Systems*, 29 (4), 1997, pp. 413-436.
[19] J. Schot, *The role of Architectural Semantics in the formal approach of Distributed Systems design*, Ph.D. thesis, University of Twente, Enschede, The Netherlands, Feb. 1992.
[20] Sun Microsystems, Inc., *Java (TM) Message Service (JMS Specification Final Release 1.1),* April 2002; http://java.sun.com/products/jms/
[21] Sun Microsystems, Inc., *J2ME Mobile Information Device Profile (MIDP)*; http://java.sun.com/products/midp/
[22] University of Twente, *The Interaction Systems Design Language (ISDL)*; http://isdl.ctit.utwente.nl/
[23] C.A. Vissers, M. van Sinderen, L. Ferreira Pires, "What makes industries believe in formal methods", *Proc. of the 13th Int'l Symp. on Protocol Specification, Testing, and Verification (PSTV XIII)*, Elsevier Science Publishers, 1993, pp. 3-26.
[24] R. Wieringa, "A survey of structured and object-oriented software specification methods and techniques," *ACM Computing Surveys*, 30 (4), 1998.
[25] World Wide Web Consortium, *SOAP Version 1.2 Part 1: Messaging Framework*, W3C Proposed Recommendation, May 2003, available at http://www.w3.org/TR/soap12-part1
[26] World Wide Web Consortium, *Web Services Description Language (WSDL) 1.1*, W3C Note, March 2001, available at http://www.w3.org/TR/wsdl