International Journal of Cooperative Information Systems

World Scientific
www.worldscientific.com

MODEL-DRIVEN DESIGN, REFINEMENT AND TRANSFORMATION OF ABSTRACT INTERACTIONS*

JOÃO PAULO A. ALMEIDA[1,2], REMCO DIJKMAN[1], LUÍS FERREIRA PIRES[1],
DICK QUARTEL[1], MARTEN VAN SINDEREN[1]

[1]*Centre for Telematics and Information Technology, University of Twente,*
P.O. Box 217, 7500 AE Enschede The Netherlands
[2]*Telematica Instituut, P.O. Box 589, 7500 AN Enschede, The Netherlands*
{j.p.andradealmeida, r.m.dijkman, l.ferreirapires, d.a.c.quartel, m.j.vansinderen}@utwente.nl

Received Day Month Day
Revised Day Month Day

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. In addition, 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. Target designs are related to source designs by conformance. We discuss how transformation and conformance can be related, such that transformations indeed preserve the characteristics prescribed by a source design.

Keywords: Model-driven design; abstract interactions; interaction refinement.

1. Introduction

In our previous work [2, 3], 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). 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.

* 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.

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.

We aim at capturing both structural and behavioural aspects of distributed applications in platform-independent models. This is in contrast with many approaches in the literature (e.g., [20, 24]), which focus on structural platform-independent models, addressing behavioural aspects only at platform-specific realization level. In these approaches, the effort invested on capturing the behaviour of an application (which amount to a large part of the distributed application effort) cannot be reused for realizations on different target 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 limitations of our approach. Section 8 discusses how transformation and conformance can be related, such that transformations indeed preserve the characteristics prescribed by a source design. Section 9 positions our work with respect to related work, including a comparison of the proposed design concepts with those concepts underlying UML and SDL. Finally, section 10 provides our conclusions and identifies some future work.

2. Model-Driven Design

In this section we characterize the model-driven design process, emphasizing the role of conformance and platform-independence in such a process. We present the notion of platform adopted in this paper and discuss how the choice of design concepts influences the level of platform-independence that can be achieved.

2.1. *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 [23]. 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 [23]).

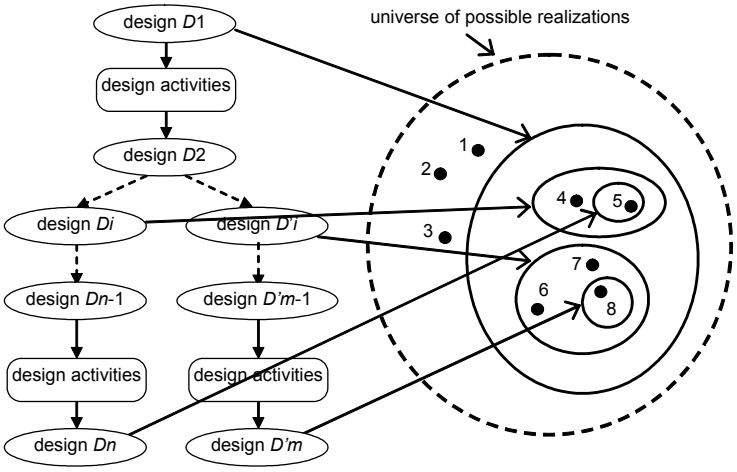


Figure 1. Reduction of realization space for designs at different levels of abstraction

Design decisions taken in a design step should meet two requirements for the design process to make progress [14]: (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 sections 5 and 8). 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.

2.2. *Platforms and platform-independence*

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. Examples of such platforms are CORBA/CCM [19] and Web Services [30]. 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 Model-Driven Architecture (MDA) [17]. 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 [2]. For example, if a platform-independent design contains application parts that interact through operation invocations (e.g., in a UML [18] 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 [19] is selected as a target platform, this characteristic can be mapped onto CORBA operation invocations. Similarly, if JMS [25] is chosen as a target platform, this characteristic can be mapped onto a pair of message exchanges [3].

2.3. *Design concepts and middleware-platform-independence*

The design concepts that a designer uses to describe an application model affect the level of platform-independence of that model. This is because each design concept represents certain design characteristics, which correspond to design decisions made (implicitly) when the concept is used.

For example, if a designer chooses to define the behaviour of each application part using a non-concurrent state-machine, then an application part can only process a single

interaction at a time. As a result, the application model excludes platform-specific implementations in which interactions may occur concurrently. This example shows that careful consideration is necessary when choosing design concepts, such that we can obtain the level of platform-independence we aim for.

The set of design concepts we require should allow the designer to exploit the spectrum of levels of platform-independence which is required to balance between two goals [2]: (i) accommodating as many target middleware platforms as possible; and (ii) defining models that can be straightforwardly implemented on concrete platforms. Goal (i) calls for highly abstract models that cannot be directly implemented on any platform, and, therefore, have to be systematically refined so that goal (ii) can be accomplished. In this paper, we discuss the concepts necessary for describing interactions at various levels of middleware platform-independence, ranging from relatively abstract (PIM) levels to more concrete ones.

3. Running Example: The Design of a Conferencing Application

We introduce a running example in the context of which the design concepts and design operations we propose can be illustrated. It consists of the design of a conferencing application, which facilitates the interaction of users residing in different hosts.

We suppose initially that 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 only 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.

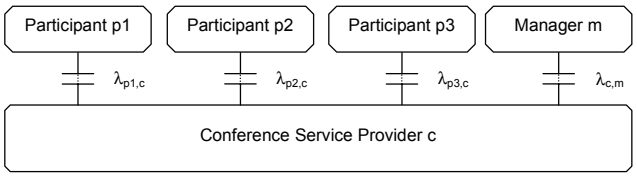


Figure 2. A snapshot of design D_0

We distinguish two basic approaches to further refine design D_0:

(1) *interaction refinement* [10], 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;

(2) *entity refinement* (called *interaction allocation and flowdown* in [29]), 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.

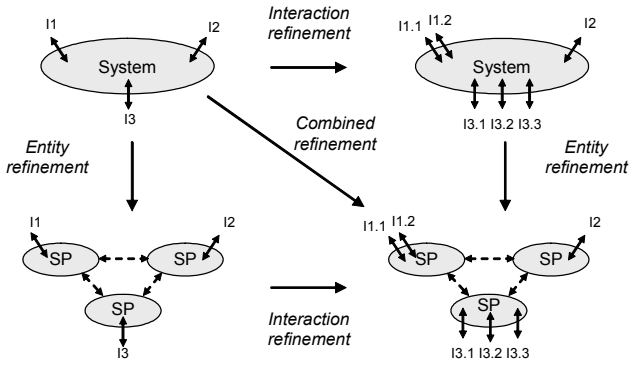


Figure 3. Approaches to system refinement [10]

We consider several alternative transformations of design D_0, according to the *interaction refinement* approach. We do not discuss entity refinement further in this paper, since this has been the subject of our previous work [3]. 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 [26]. The abstract platform used in D_4 facilitates the realization of this design using the Java Message Service (JMS) [25] or the CORBA Notification Service.

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

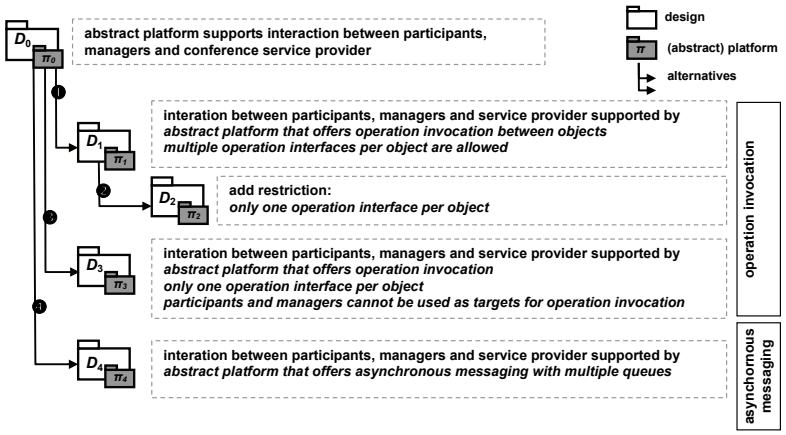


Figure 4. Alternative design steps

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.

4. Concepts for Abstract Platform Design

In this section we generalize the alternative design steps of section 3 to derive requirements for design concepts at various levels of platform-independence. We also propose some basic design concepts that fulfil the requirements.

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. An abstract interaction concept should abstract from details of interaction mechanisms and allow the designer to use any mechanism for the realization of a design (such as operation invocation and asynchronous messaging). Therefore, we propose an interaction concept that only represents: (i) the identity of the interaction; the successful occurrence of the interaction; (ii) 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, (iii) the direction in which the information flows. Such a concept abstracts from the roles that the interacting parties play in the interaction (e.g., initiator or responder) and other aspects of interaction mechanisms that are deferred to a later stage of the design process (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: (i) the identity of the interface; (ii) the interactions that are supported by the interface, as well as the relations between these interactions; and (iii) the party that interacts via the interface. Such a concept abstracts from: (i) any constraints on the interaction mechanisms that are available at the interface (e.g., only remote procedure calls can occur at this interface); (ii) 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); and, (iii) 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*

The basic design concepts presented in Figure 5 satisfy the requirements defined in section 4.1, by defining abstract interaction and interface concepts. These concepts are adapted versions of the RM-ODP basic modelling concepts [13] as explained in [9].

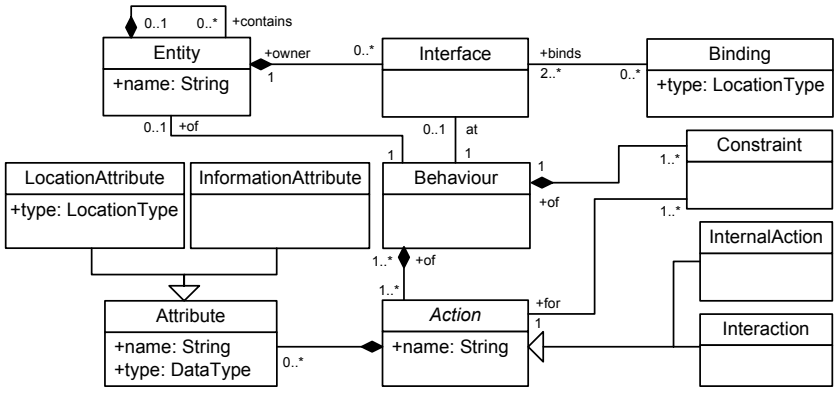


Figure 5. Conceptual model

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. A binding does not represent something *in between* the interfaces. The bound interfaces themselves constitute the mechanism[†].

[†] In this paper, we consider a system given a certain configuration of entities, interfaces and bindings, and we do not address modification of the system structure.

Entities have a behaviour according to which they perform actions. Interfaces also have behaviours, which partially determine the behaviour of an entity and 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, namely, *value passing*, *value checking* and *value generation* [22]. 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 2 presents a snapshot of the structural aspects of D_0 in terms of the basic concepts described above. An entity has been represented by a rectangle with cut-off corners that contains the entity's name. An interface is represented by a "T" connected to its owning entity. A binding is represented by a dashed line that connects the bound interfaces. Bindings are annotated with their location.

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 2). 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 6 represents 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. Furthermore, 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.

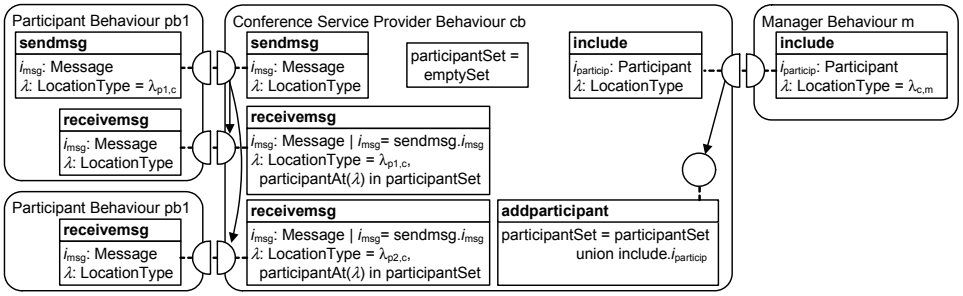


Figure 6. Conference system behaviour

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 a behaviour. An interaction contribution is represented by a semi-circle drawn on the border of a behaviour. 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. 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). In this paper we do not discuss the precise way to represent constraints. We refer to [22] for more information about this aspect.

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 below, 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) cannot 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 cannot be refined into an arbitrary set of actions and constraints, because the refined behaviour must preserve the characteristics that the original behaviour prescribes. [21] 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.

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 mechanism 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 server to respond to a request and an interaction for a client to receive the response to the request. Table 1 shows a transformation that refines an interaction into multiple interactions forming a remote invocation.

Table 1. 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 cannot directly support the binding. 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. 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. Interactions that occur at a certain binding should occur at locations introduced by bindings or entities that replace it.

5.4. *Binding decomposition example*

We obtain design D_1 from D_0 in two steps. Table 2 shows the transformation used in the first step, in which the bindings from D_0 are decomposed into multiple entities.

Table 2. 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 7 illustrates this decomposition step graphically.

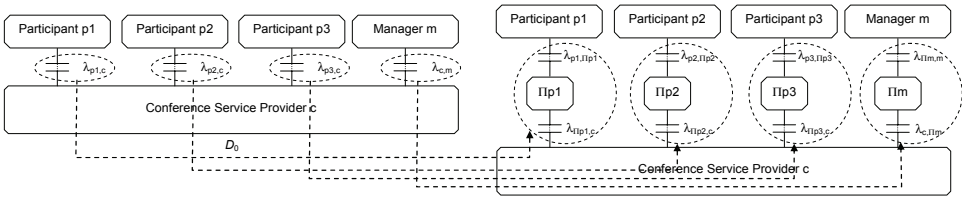


Figure 7. Action refinement and binding decomposition applied to D_0

The interactions that occurred at the original binding are refined according to the rule from Table 1. 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 7). 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[‡]. In this refinement, the name of the operation is sendmsg (not to be confused with the sendmsg interaction from Figure 6) 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; requesting the operation to be invoked. 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 7). 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 8 represents part of the refined behaviour. However, it only shows one participant. The behaviour of the abstract platform is such that it can accept invocation_req interactions at both the binding with the participant and the binding with the conference service provider. This is because it does not restrict the location λ at which this interaction can take place (i.e., there is no constraint for the location attribute of invocation_req). This means that the invocation_req interaction contribution in fact is part of two interactions: one interaction between the abstract platform and the conference service provider, and one interaction between the abstract platform and the participant. Upon engaging in a invocation_req in a certain binding, the abstract platform causes an invocation_ind to occur at the other ("opposite") binding. The behaviour of the conference service provider ensures that after engaging in a invocation_ind interaction in which the sendmsg operation is invoked, the conference service provider enables an invocation_req, in which the receivemsg operation is invoked.

In Figure 8, invocation_ind with a value of sendmsg for i_{op} is a final action for sendmsg from Figure 6. Similarly, invocation_ind with a value of receivemsg for i_{op} is a final action for receivemsg from Figure 6. 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 6 that receivemsg is caused by sendmsg is also enforced by the final actions for receivemsg and sendmsg from Figure 8.

[‡] 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.

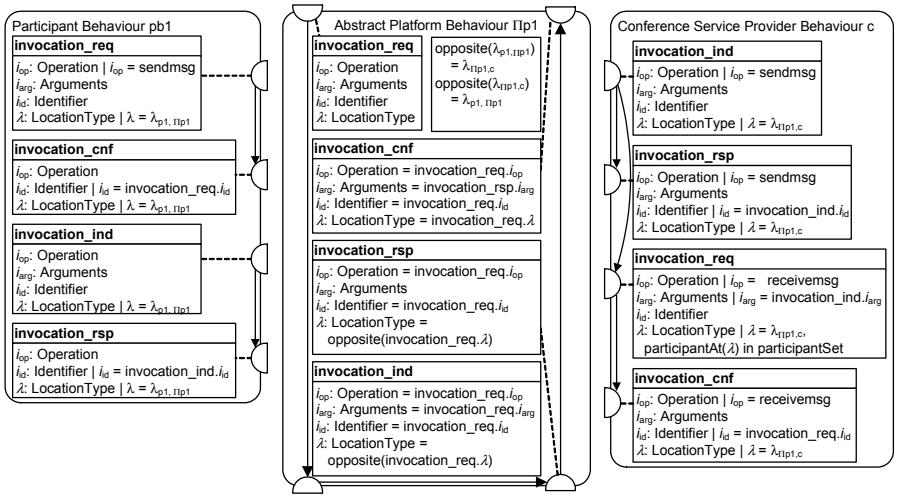


Figure 8. Refined behaviour

In the design depicted in Figure 7 and Figure 8 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 9 illustrates this.

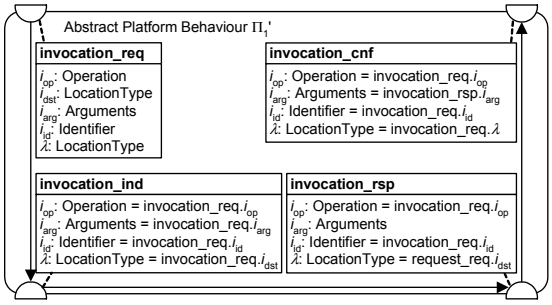


Figure 9. Invocation target as attribute i_{dst}

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. The resulting entity has all the

bindings that the original entities had. Similarly, the resulting entity carries all the behaviours of the original entities.

5.6. *Entity merging example*

Figure 10 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.

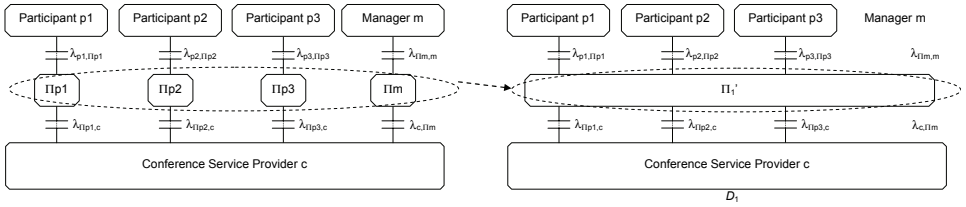


Figure 10. Entity merging to obtain D_1

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 cannot be distinguished anymore. For example, if a binding λ replaces a set of bindings λ_i, 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 11.

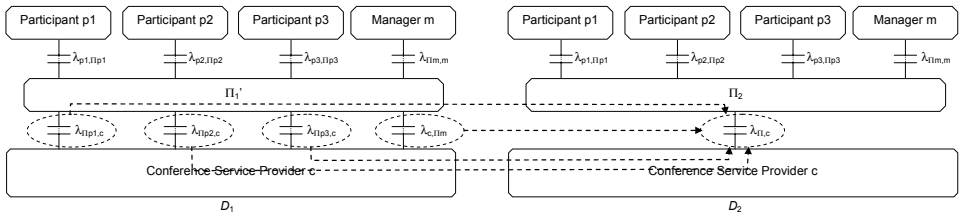


Figure 11. Binding and interface merging applied to D_1, resulting in D_2

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 $\lambda_{\Pi,c}$ that 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, i.e., each pattern of concepts in the design corresponds to a construct that is provided directly by the platform. 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).

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. In D_3 and D_4 *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:

- an 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;
- an 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;
- an invocation_rsp 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;

- an invocation_cnf interaction, which occurs at the same binding at which the invocation_req interaction occur. This interaction follows the occurrence of the invocation_rsp 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 $\lambda_{\text{p}n,\Pi\text{p}n}$ 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 $\lambda_{\Pi\text{p}n,\text{c}}$ 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 [21].

7. Discussion

In this section, we discuss some issues related to the use of the design concepts proposed in section 4.

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.

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.

8. Conformance and Transformation

In this section we explain the relation between conformance and transformation. We also show an example that illustrates this relation.

8.1. *Relation between conformance and transformation*

Conformance rules and (non-parameterized) transformation specifications can be regarded as two extreme approaches in relating source and target designs from the perspective of design freedom for the target design. Conformance rules determine the minimum to be preserved in a design step (hence maximum freedom for target design without losing design decisions in the source design) and transformation specifications determine the maximum that can be prescribed in a design step (hence resulting in a specific target design, minimum freedom). This is illustrated in Figure 12.

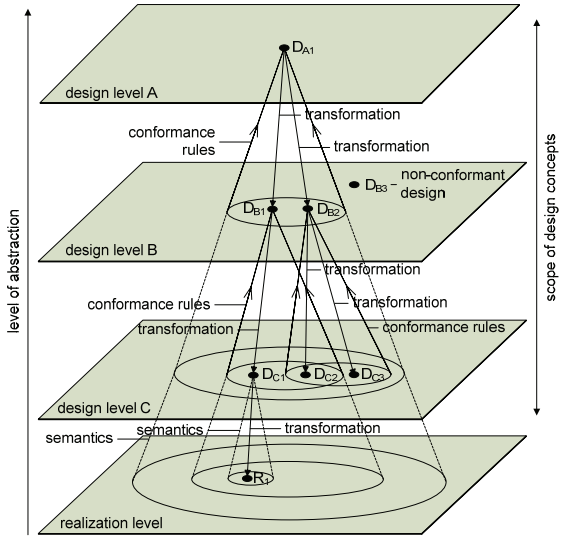
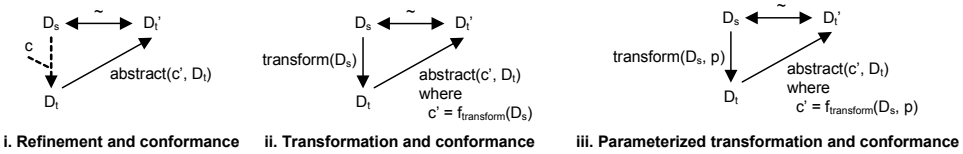


Figure 12. Conformance and transformation

Conformance rules determine implicitly the set of target designs which conform to a source design. These rules are depicted as a "spotlight"; a source design "illuminates" a target area in the target design level. In contrast, a (non-parameterized and deterministic) transformation specification is depicted as an arrow from a source to a target design.

In order to ensure that the characteristics that the source design prescribes are preserved in the target design conformance assessment should be conducted. This is illustrated by the arrows pointing towards the source design in Figure 12.

Conformance assessment can be carried out as illustrated in Figure 13.i. This figure illustrates that during the construction of a target design D_t from a source design D_s, characteristics *c* are inserted as a consequence of design decisions that are made. To check the conformance of the target design to the source design, we have to abstract from these characteristics, resulting in an abstracted target design D_t'. Subsequently, we must check if the abstracted target design is equivalent (~) to the source design, according to a chosen notion of equivalence. To abstract from inserted characteristics *c*, we do not necessarily need to know the details of these characteristics. Therefore, the 'abstract' operator refers to a set of characteristics *c*'. For example, if, during the construction of a target design, an interaction is inserted, we only need to know which interaction we must abstract from during the conformance assessment. We do not need to know details about this interaction.



i. Refinement and conformance **ii. Transformation and conformance** **iii. Parameterized transformation and conformance**

Figure 13. Conformance and transformation

A transformation from a source design D_s into a target design D_t is a special case of constructing a target model. In this case the target model and the inserted characteristics are completely determined by the transformation rules and the source design D_s. Hence, the characteristics *c*' that we must abstract from during the conformance assessment can be automatically determined by a function $f_{transform}$ on the source design D_s. This function must be defined as a complement of the transformation. If such a function exists, we can automatically check the conformance of the target model to the source model. Figure 13.ii illustrates this case. Figure 13.iii illustrates the case in which the transformation is parameterized. For parameterized transformations, design decisions taken in the design step are determined by the transformation rules, the source design and the values of the parameters. Hence, *c*' can be determined by a function $f_{transform}$ on the source design D_s and the parameter values *p*.

An approach to handle conformance assessment is to define transformations that always produce conformant target designs. The benefit of this approach is that conformance assessment activities do not have be performed (manually) for each application of the transformation. This is particularly beneficial in iterative design

approaches in which transformations are re-applied frequently to cope with changes in source models.

To show that a model transformation guarantees conformance, we must prove that for each source design D_s:

$$\text{abstract}(\text{transform}(D_s), f_{\text{transform}}(D_s)) \sim D_s.$$

For a parameterized transformation, we must prove that for each source design D_s and each admissible parameter value for D_s:

$$\text{abstract}(\text{transform}(D_s), f_{\text{transform}}(D_s, p)) \sim D_s.$$

8.2. *Example*

In this section, we illustrate the approaches to conformance assessment discussed above. As an example of assessing conformance of a transformed design, we consider the source design D_s from Figure 14.i as a starting point. To this design, we apply the transformation described in Table 1. This results in a target design D_t in which application parts interact through a request/response pattern, as depicted in Figure 14.ii. The source and target designs are adapted versions of the designs presented in Figure 6 and Figure 8.

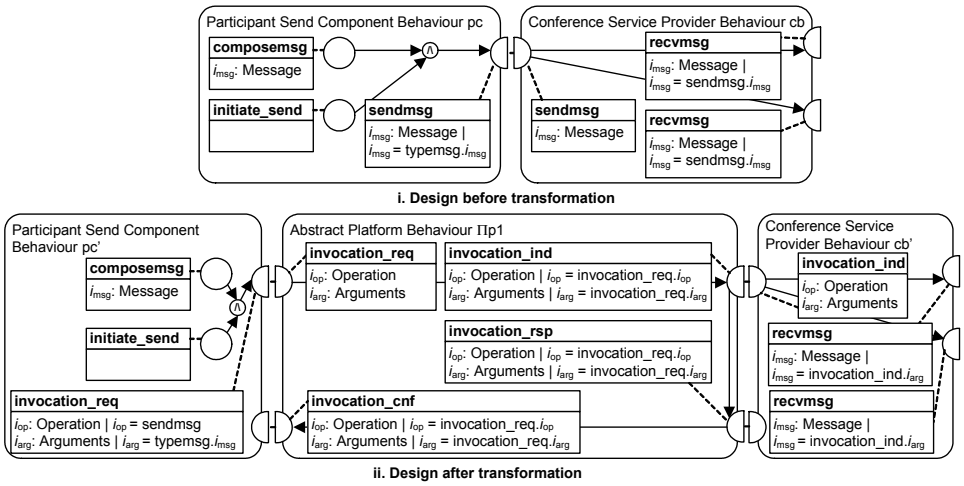


Figure 14. Example of conformance and transformation

To assess the conformance of D_t to D_s, we apply the conformance assessment approach from Figure 13.ii.

During the transformation, the sendmsg interaction is refined by the invocation_req, invocation_ind, invocation_rsp and invocation_cnf interactions and the abstract platform behaviour Πp1. To perform conformance assessment, we must construct a function $f_{transform}$ that yields the characteristics c' that are inserted into the source design. We can do that by observing that the invocation_ind interaction is the final action, the completion of which corresponds to the completion of the sendmsg interaction. The other interactions

are considered inserted interactions. invocation_ind establishes two results, i_{op} and i_{arg}, where i_{arg} corresponds to the result i_{msg} of sendmsg and i_{op} is inserted. Finally, the behaviour Πp1 of the abstract platform is inserted. Hence:

$$f_{transform}(D_s) = \{ \text{invocation\_req, invocation\_rsp, invocation\_cnf, } i_{op}, \Pi p1\}$$

When abstracting from invocation_req, invocation_rsp and invocation_cnf, we must rewrite the relations that invocation_ind has with these actions. Also, we must rewrite the constraints on invocation_ind's attributes that refer to attributes of inserted interactions. To rewrite the relations, note that the occurrence of invocation_ind is caused by the occurrence of invocation_req, while the occurrence of invocation_req is caused by the occurrence of composemsg and initiate_send. Hence, when we abstract from invocation_req, the occurrence of invocation_ind is caused by the occurrence of composemsg and initiate_send. Similarly, to rewrite the constraints on the attributes of invocation_ind, note that invocation_ind.i_{arg} = invocation_req.i_{msg} and invocation_req.i_{arg} = composemsg.i_{msg}. Hence, after abstracting from invocation_req, invocation_ind must enforce that invocation_ind.i_{arg} = composemsg.i_{msg}.

When abstracting from Πp1, we must re-assign invocation_ind to some behaviour other than Πp1. We assign an abstraction of Participant Send Component Behaviour pc' (pc'') with the responsibility of enforcing that invocation_ind only occurs after composemsg and initiate_send have occurred. We do that, because originally Πp1 and pc' enforced that condition. Hence, when we abstract from Πp1 only pc" enforces it. Similarly, we assign pc" the responsibility of enforcing that i_{arg} = composemsg.i_{msg}. Abstracting from the inserted characteristics yields an interaction invocation_ind that establishes the information value i_{arg}, for which invocation_ind.i_{arg} = composemsg.i_{msg}. This interaction is equivalent to the sendmsg interaction from Figure 14.i minus naming differences. Hence, the behaviour after abstracting from inserted characteristics is equivalent to the behaviour shown in Figure 14.i. Therefore, the transformation yields a behaviour that conforms to D_s.

Our approach to verify conformance of behaviours is described precisely in [8, 21]. As an example of proving that a transformation always yields a target design that conforms to the source design, we prove that the transformation from Table 1 has this property under certain conditions. The proof focuses on conformance with respect to relations between actions.

To construct the proof, we argue that, in any source design D_s, we can group the relations that an interaction i can have into four groups, as illustrated by Figure 15.i:

- A_1 is the group of actions on which the occurrence of i depends in b_1, including actions that have a choice relation with i (an action has a choice relation with i, if either that action or i can occur, but not both);
- A_2 is the group of actions on which the occurrence of i depends in b_2;
- A_3 is the group of actions that depend on the occurrence of i in b_1, excluding actions that have a choice relation with i;
- A_4 is the group of actions that depend on the occurrence of i in b_2.

Figure 15.ii illustrates the transformation *transform_table_2*(D_s) that transforms a source design D_s, using the rules from Table 1. The transformation is undefined for disabling relations in b_1. Therefore, the transformation can only be applied if the condition is

satisfied that b_1 does not contain any disabling relation. An action a_1 disables another action a_2, if after a_1's occurrence, a_2 can not occur anymore. For example, the occurrence of a 'cancel' action, disables the occurrence of all actions in a 'sales' process.

In Figure 15.ii, i_1, i_3 and i_4 are inserted interactions and *ib* is an inserted behaviour. i_2 is the final interaction for *i*. Hence:

$$f_{transform\_table2}(D_s) = \{i_1, i_3, i_4, ib\}.$$

To prove that the transformation always yields a conformant target design, we prove that, after abstraction from these inserted characteristics, the transformed design is equivalent to the source design. The inserted interactions do not affect the relations that i_2 has with actions in A_2 and A_3. Hence, after abstraction from the inserted interactions, these relations do not change. The occurrence i_2 depends on the occurrence of i_1, which depends on the occurrence of actions in A_1. Hence, after abstracting from i_1, the occurrence of i_2 depends on the occurrence of actions in A_1. The occurrence of actions in A_4 depends on the occurrence of i_4, which depends on the occurrence of i_3, which depends on the occurrence of i_2. Hence, after abstracting from i_3 and i_4, the occurrence of actions in A_4 depends on the occurrence of i_2. Based on these observations, Figure 15.iii shows the transformed design after abstracting from the inserted interactions and the inserted behaviour. In terms of Figure 13.ii:

$$\text{abstract}(\text{transform\_table\_2}(D_s), f_{transform\_table\_2}(D_s)).$$

The behaviour from Figure 15.iii is equivalent with the source behaviour. Hence, the transformation from Table 1 always yields a target design that conforms to the source design with respect to relations between actions. However, the condition applies that the source design can not contain actions, of which the occurrence disables the occurrence of *i* on the 'client-side' of the RPC. In terms of Figure 13.ii, for any source design D_s for which this condition holds:

$$\text{abstract}(\text{transform\_table\_2}(D_s), f_{transform\_table\_2}(D_s)) \sim D_s.$$

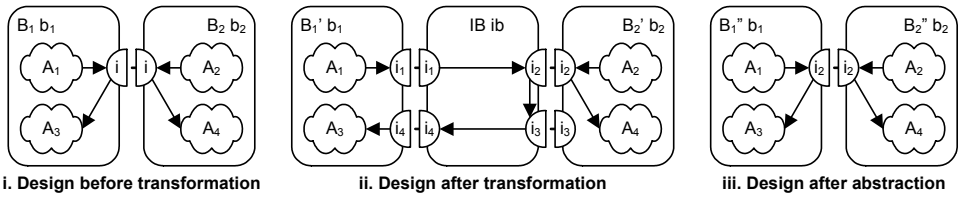


Figure 15. Conformant transformation of causal relations

9. Related Work

In this section we compare the abstract interaction concept we adopt with the interaction concepts underlying UML [18] and SDL [12] and discuss how our work on design operations and transformations is related to previous efforts documented in the literature.

9.1. *Interaction concepts underlying UML and SDL*

Popular modelling languages, such as UML and SDL use basic interaction concepts that are derived from operation invocation and message passing mechanisms. These concepts limit the level of platform independence that can be achieved, because of certain design characteristics they represent and the design decisions that these characteristics imply. We explain three such implicit design decisions and discuss their limiting effect on the level of platform independence.

Firstly, the use of message passing and remote procedure call concepts in platform-independent designs hinders the use of interaction patterns that deviate from message passing and remote procedure call, such as, e.g., interactions with transactional properties or interactions for group communication. This is because such interactions have to be represented by a particular composition of messages or remote procedure calls. An example that illustrates this problem is the use of remote procedure calls to define a notification service to support multicast interactions (similarly to the notification service which is defined in CORBA using remote procedure calls in IDL). In this case, multicast interactions are represented as multiple remote procedure calls to the notification service. This (i) unnecessarily expands a design, and (ii) favours implementation of the (multicast-) interaction in terms of the particular composition of procedure calls used in the design. An abstract interaction with the participation of multiple parties is, in this case, a more adequate abstraction to preserve freedom of implementation.

Secondly, operation invocation and message passing concepts represent both the direction in which information flows and the initiating and responding roles for an interaction. Therefore, they force a designer to prescribe the direction of an interaction and roles in an interaction at all levels of platform-independence. This, for example, does not allow a designer to defer the decision of 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 have observed 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 processing and memory resources. Installing the server-side part of an ORB is not required when the designer chooses for a polling mechanism. An abstract interaction such as the one proposed in this paper allows a higher level of platform-independence by supporting the designer in deferring design decisions that are platform-specific such as the choice of a callback and polling mechanism as discussed in the example above.

Thirdly, 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 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.

An attempt to mitigate this third issue is the use of semantic variation points in UML. The UML specification 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." [18] 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 application models that behave differently. For example, a design in which requests are re-ordered during transportation is different from a design in which they are not. If a designer does not make this choice explicit other designers and simulation, verification and validation tools may draw the wrong conclusions about the design. On the other hand, is a designer does make a choice this has consequences for choices later on in the process. Strictly speaking, this may even lead to situations in which the designer would have to implement a mechanism to re-order requests during transportation. 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 and compulsory choice should not be confused with abstraction).

Another attempt to mitigate some of the limitations we have discussed in this section (other than changing the set of design concepts) is to interpret behavioural specifications loosely. In this case, designers choose to ignore certain characteristics that are implied by (platform-independent) models to be able to use the models as a starting point for realizations in different platforms. We do not consider this approach since it relies on the lack of precision for source models, which severely restricts their usefulness for model transformation, automated testing, validation and simulation.

9.2. *Design operations and transformation*

Design transformations in which implementation constraints are incorporated have been proposed earlier, for example, in the LOTOSphere [5] project. Some of the design operations we have presented here have been inspired by the transformations described in [23]. 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 [23] 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.

Some model-driven approaches simplify the issues of transformation and conformance by restricting the use of the modelling language at the source level in such a way that only combinations of concepts that can be directly mapped into concepts at the target level can be used (e.g., [11]). While these approaches guarantee the feasibility of the realization in a particular platform, this comes at the cost of introducing platform constraints at the source level, reducing the level of platform-independence that can be achieved. We believe these approaches are only suitable for models at relatively low levels of platform-independence.

We approach interaction refinement from the perspective of architectural design. Several authors have approached interaction refinement from a pure formal perspective (e.g., [6, 7]). 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 [28]). For example, in [7] interactive systems communicate asynchronously via unbounded FIFO channels.

In [15], a methodology is proposed for the step-wise refinement of heterogeneous software architectures. Refinement patterns are used that represent solutions to standard architectural design problems. These refinement patterns are compositional and can be proved correct in isolation. A small number of patterns for refining components, interfaces and connectors are defined.

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 techniques are neutral with respect to the abstraction criteria and design concepts used for platform-independent design. Such work is complementary to the work presented in this paper, since the design concepts and operations we have defined can be used to derive model transformation specifications that could be implemented by tools.

10. Conclusions

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.

Many model-driven design approaches address solely the structural aspects of an application's design. In these cases, model transformations often consist of isomorphic relations between source and target structural models. For these kinds of relations between source and target models, conformance can be guaranteed by construction. However, this comes at the cost of abstraction and platform-independence, with application models serving the purpose of visualizing a (platform-specific) realization. As we have shown in this paper, the incorporation of platform constraints in a design step requires a transformation of both behavioural and structural aspects of the design. These transformations are rarely isomorphic and therefore, more attention is required to ensure

that properties defined at a particular level of abstraction are preserved during transformation.

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 interactions 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) [27] 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 that modify the system structure during execution. The application of the interaction refinement operations presented here when considering the dynamic creation and destruction of entities, interfaces and bindings remains to be investigated.

We have implemented tool support for a specific transformation that applies (inter)action refinement as presented in this paper. This tool is integrated with an ISDL simulator developed at the University of Twente. This experience has been reported in [4], which also reports on the use of the design concepts discussed here for the design of context-aware services. We intend to develop tool support that implements the design operations presented in this paper in terms of (semi-) automated model transformations, which would contribute for a broader application and validation of our approach.

Further work should investigate both conformance and transformation within the same transformation framework, possibly using the same techniques and tools for model transformation and for capturing and enforcing conformance rules. We believe this is feasible by regarding both transformation and conformance as relations ([1, 16]).

References

1. D. Akehurst, S. Kent, O. Patrascoiu, A relational approach to defining and implementing transformations between metamodels, *Software and Systems Modeling*, vol. 2. no. 4 (Springer-Verlag, 2003), pp. 215–239.
2. J.P.A. Almeida, R. Dijkman, M. van Sinderen and L. Ferreira Pires, On the Notion of Abstract Platform in MDA Development, in *Proc. 8th IEEE Int'l Conf. on Enterprise Distributed Object Computing (EDOC 2004)* (IEEE CS Press, 2004), pp. 253–263.
3. 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, in *Proc. 7th IEEE Int'l Conf. on Enterprise Distributed Object Computing (EDOC 2003)* (IEEE CS Press, 2003), pp. 112–134.
4. J.P.A. Almeida, *Model-Driven Design of Distributed Applications*, CTIT Ph.D.-Thesis Series, No. 06-85, Telematica Instituut Fundamental Research Series, No. 018 (2006).
5. T. Bolognesi, J. van de Lagemaat, and C. Vissers, eds., *LOTOSphere: Software Development with LOTOS* (Kluwer Academic Publishers, 1995).

6. 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, Vol. 494 (Springer-Verlag, 1991), pp. 297–312.
7. M. Broy, (Inter-)action refinement: The easy way, *Program Design Calculi*, Springer NATO ASI Series, Series F: Computer and System Sci. 118 (Springer-Verlag, 1993), pp. 121–158.
8. R.M. Dijkman, *Consistency in Multi-Viewpoint Architectural Design*, CTIT Ph.D.-Thesis Series, No. 06-80, Telematica Instituut Fundamental Research Series, No. 017 (2006).
9. R.M. Dijkman, D. Quartel, L. Ferreira Pires, M. van Sinderen, A Rigorous Approach to Relate the RM-ODP Enterprise and Computational Viewpoint, in *Proc. 8th IEEE Int'l Conf. on Enterprise Distributed Object Computing (EDOC 2004)* (IEEE CS Press, 2004), pp. 187–200.
10. C.R.G. de Farias, *Architectural design of groupware systems: a component-based approach*, Ph.D. thesis, (University of Twente, The Netherlands, 2002).
11. IBM, *Draft UML 1.4 Profile for Automated Business Processes with a mapping to BPEL 1.0, Version 1.1* (2003), http://ibm.com/developerworks/rational/library/4593.html
12. ITU-T, *Recommendation Z.100 – CCITT Specification and Description Language* (2002).
13. ITU-T / ISO, *Open Distributed Processing - Reference Model – All Parts*, ITU-T X.901-4 | ISO/IEC 10746-1 to 10746-4 (1995).
14. H. Kremer, *Protocol Implementation: Bridging the gap between Architecture and Realization*, Ph.D. thesis (University of Twente, The Netherlands, 1995).
15. M. Moriconi, X. Qian, and R.A. Riemenschneider, Correct Architecture Refinement, *IEEE Transactions on Software Engineering*, 21(4) (IEEE CS Press, 2005).
16. Object Management Group, *MOF QVT Final Adopted Specification*, ptc/05-11-01 (2005).
17. Object Management Group, *MDA-Guide, V1.0.1*, omg/03-06-01 (2003).
18. Object Management Group, *UML 2.0 Superstructure*, ptc/03-08-02 (2003).
19. Object Management Group, *Common Object Request Broker Architecture: Core Specification, Version 3.0*, formal/02-12-06 (2002).
20. O. Patrascoiu, Mapping EDOC to Web Services using YATL, in *Proc. 8th IEEE Int'l Conf. on Enterprise Distributed Object Computing (EDOC 2004)* (IEEE CS Press, 2004) pp. 286– 297.
21. 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).
22. 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) (Elsevier Science Publishers, 1997) pp. 413–436.
23. J. Schot, *The role of Architectural Semantics in the formal approach of Distributed Systems design*, Ph.D. thesis (University of Twente, The Netherlands, 1992).
24. R. Silaghi, F. Fondement, and A. Strohmeier, Towards an MDA-Oriented UML Profile for Distribution, in *Proc. 8th IEEE Int'l Conf. on Enterprise Distributed Object Computing (EDOC 2004)* (IEEE CS Press, 2004) pp.227–239.
25. Sun Microsystems, Inc., *Java (TM) Message Service (JMS Specification Final Release 1.1)*, (2002), http://java.sun.com/products/jms/
26. Sun Microsystems, Inc., *JSR-000037 Mobile Information Device Profile (MIDP)*, (2000), http://jcp.org/aboutJava/communityprocess/final/jsr037/
27. *The Interaction Systems Design Language (ISDL)*, http://isdl.ctit.utwente.nl/
28. C.A. Vissers, M. van Sinderen, and L. Ferreira Pires, What makes industries believe in formal methods, in *Proc. of the 13th Int'l Symp. on Protocol Specification, Testing, and Verification (PSTV XIII)* (Elsevier Science Publishers, 1993) pp. 3–26.
29. R. Wieringa, A survey of structured and object-oriented software specification methods and techniques, *ACM Computing Surveys*, 30 (4) (ACM Press, 1998).
30. World Wide Web Consortium, *Web Services Architecture*, http://www.w3.org/TR/ws-arch/