Platform-Independent Modelling in MDA: Supporting Abstract Platforms

João Paulo Almeida, Remco Dijkman, Marten van Sinderen, Luís Ferreira Pires

Centre for Telematics and Information Technology, University of Twente
PO Box 217, 7500AE, Enschede, The Netherlands
{ almeida, dijkman, sinderen, pires } @ cs.utwente.nl

Abstract. An MDA-based design approach should be able to accommodate designs at different levels of platform-independence. We have previously proposed a design approach [2], which allows these levels to be identified. An important feature of this approach is the notion of abstract platform. An abstract platform is determined by considering the platform characteristics that are relevant for applications at a certain level of platform-independence as well as the various design goals. In this paper, we discuss how our design approach can be supported using the MDA standards UML 2.0 and MOF 2.0. Since our methodological framework is based on the notion of abstract platform, we pay particular attention to the representation of abstract platforms and the language requirements to specify abstract platforms.

1 Introduction

A current trend in the development of distributed applications is to separate their technology-independent and technology-specific aspects, by describing them in separate models. The most prominent example of this trend is the Model-Driven Architecture (MDA) [15], [18]. A common pattern in MDA development is to define a platform-independent model (PIM) of a distributed application, and to apply (parameterised) transformations to this PIM to obtain one or more platform-specific models (PSMs). The main benefit of this approach stems from the possibility to derive different alternative PSMs from the same PIM depending on the target platform, and to partially automate the model transformation process and the realization of the distributed application on specific target platforms.

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

trajectory, how these models are related, and which criteria should be used for their application. Some of the results of these investigations have been presented earlier in [2], where we have proposed an MDA design trajectory that accommodates designs at different levels of platform-independence.

An architectural concept that plays an important role in this approach is that of *abstract platform*. An abstract platform defines an acceptable or, to some extent, ideal platform from an application developer's point of view; it represents the platform support that is assumed by the application developer at some point of (the platform-independent phase of) the design trajectory. Alternatively, an abstract platform defines characteristics that must have proper mappings onto the set of concrete target platforms that are considered for an MDA design process, thereby defining the level of platform-independence for this particular process. Defining an abstract platform forces a designer to address two conflicting goals: (i) to achieve platform-independence, and (ii) to reduce the size of the design space explored for platform-specific realization.

Any design approach that is intended to be successfully applied in practice should be supported by suitable design concepts in suitable design languages. In this paper, we present some methodological guidelines for platform-independent design and define requirements for design languages intended to support platform-independent design. Since our methodological framework is based on the notion of abstract platform, we pay particular attention to the representation of abstract platforms and the language requirements to specify them. We discuss how the architectural concept of abstract platform can be supported in UML 2.0 [23] and MOF 2.0 [19].

This paper is further structured as follows: Section 2 provides some background on the concept of abstract platform; Section 3 discusses how abstract platforms relate to design languages; Section 4 discusses how abstract platforms can be represented in UML 2.0 and MOF 2.0; Section 5 presents examples of abstract platforms and their representations; Section 6 discusses limitations of UML 2.0 with respect to the representation of abstract platforms; Section 7 positions our work with respect to related work. Finally, Section 8 presents our conclusions and outlines future work.

2 Abstract Platforms

Platform-independence is a quality of a model that relates to the extent to which the model abstracts from the characteristics of particular technology platforms. In order to refer to platform-independent or platform-specific models, one must define what a platform is. The following rather general definition of platform can be found in [18] (page 2-3): "a platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns". This paper concentrates on platforms that correspond to some middleware technology supporting operation invocation and asynchronous message exchange, such as CORBA/CCM [16], .NET [13] and Web Services [28], [29].

When pursuing platform-independence, one could strive for PIMs that are neutral with respect to all different classes of middleware platforms. This is possible for models in which the characteristics of the supporting technological infrastructure are

irrelevant, such as, e.g., conceptual domain models [4] and RM-ODP Enterprise Viewpoint models [9], which can be considered as Computation Independent Models [18]. However, along a development trajectory, when system architecture is captured, some platform characteristics become relevant, and different sets of platform-independent modelling concepts may be used, each of which being adequate only with respect to specific classes of target middleware platforms. This leads to the observation that platform-independence is not a binary quality of models; instead, a distributed application can be described at several levels of platform-independence. The level of platform-independence of a model must be carefully identified. We propose to make this identification an explicit step in MDA development. The notion of abstract platform, as proposed initially in [2], supports a designer in this step.

An abstract platform is determined by the platform characteristics that are relevant for applications at a certain platform-independent level. For example, if a platform-independent design contains application parts that interact through operation invocations, then operation invocation is a characteristic of the abstract platform. Capabilities of a concrete platform are used during platform-specific realization to support this characteristic of the abstract platform. For example, if CORBA is selected as a target platform, this characteristic can be mapped onto CORBA operation invocations.

The PIM of a distributed application depends on an abstract platform model, in the same way as the PSM depends on a (concrete) platform model (see Figure 1). Given the PIM of an application and an abstract platform model, we distinguish two contrasting extreme approaches to proceed with platform-specific realization:

1. *Adjust the concrete platform*, so that it corresponds directly to the abstract platform.
2. *Adjust the (scope of the) application during platform-specific realization, such that* the requirements specified at platform-independent level are preserved and the platform-specific application model can be composed with the target platform model.

In approach 1, the boundary between abstract platform and platform-independent application model is preserved during platform-specific realization. This implies the introduction of some platform-specific *abstract platform logic* to be composed with the concrete target platform. The nature of this composition depends on the particular requirements for the abstract platform. It may be possible to implement abstract platform logic on top of the concrete platform. Nevertheless, this composition may also imply the introduction of platform-specific (e.g., QoS) mechanisms, possibly defined in terms of internal components of the concrete platform. Extension in a non-intrusive manner is often the preferred way to adjust the concrete platform. Techniques that can be used for non-intrusive extension include interceptors [16], aspect-oriented programming and composition filters [5].

Approach 2 may imply the introduction of (e.g., QoS) mechanisms in the platform-specific design of the application. This approach may be suitable in case it is impossible to adjust the concrete target platform, e.g., due to the lack of extension mechanisms or the cost implications of these adjustments.

Figure 1 illustrates these approaches to platform-specific realization.

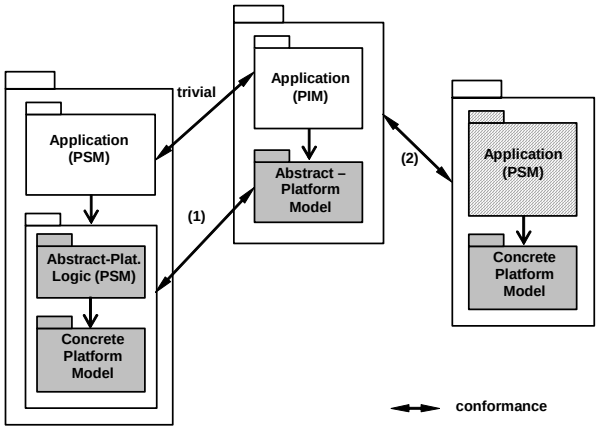

Fig. 1. Alternative approaches to platform-specific realization

Both approaches allow us to target different concrete platforms from the same platform-independent model, with different quality characteristics [2]. Approach 1 can be generalized as a recursive application of service definition (external perspective) and the service's internal design, resulting in a hierarchy of abstract platforms and a concrete target platform. At each step of the recursion, both approaches to realization can be chosen.

3 Design Languages

Designs must be supported by suitable design concepts and represented using suitable design languages. In an MDA design trajectory, several design languages may be used, e.g., to produce models at different levels of abstraction. Alternatively, a single "broad spectrum" design language [6] may be used. The design language adopted for a design has an important role in defining characteristics of an abstract platform assumed for the design.

In an MDA-based development trajectory, we may apply the *implicit abstract platform definition* approach, in which the characteristics of an abstract platform are implied by the set of design concepts used for describing the platform-independent model of a distributed application. These concepts are often inherited from the adopted modelling language. For example, the exchange of "signals" between "agents" in SDL [10] may be considered to define an abstract platform that supports reliable asynchronous message exchange. The restricted use of particular constructs in a design language or the use of certain modelling styles can serve as a means to select subsets of a language's design concepts.

Instead of implying an abstract platform definition from the adopted set of design concepts for platform-independent modelling, it may be useful or even necessary to define the characteristics of an abstract platform explicitly, resulting in one or more separate and reusable design artefacts. We call this approach *explicit abstract*

platform definition. During platform-independent modelling, parts of a pre-defined abstract platform model may be composed with the model of the distributed application. For example, although group communication is not a primitive design concept of UML 2.0, it is possible to specify the behaviour of a group communication sub-system using UML 2.0. This sub-system is then re-used in the design of a distributed application. Other examples of pre-defined artefacts that may be included in abstract platforms are the ODP trader [8] and the OMG pervasive services [18] (yet to be defined). The set of design concepts of a design language is still relevant in this approach, since the distributed application and the abstract platform model are described in the language.

In both the implicit and explicit abstract platform definition approaches, there is some overlap between language characteristics and abstract platform characteristics. This leads to the formulation of an important requirement for a design language to support platform-independent design: *the concepts underlying the design language should be precisely defined, so that the characteristics of the abstract platform can be unambiguously derived from these concepts.* This is important for at least two reasons: (1) designers need to know the characteristics of the abstract platform when defining platform-independent models of an application; and (2) abstract platforms are a starting point for platform-specific realization.

Furthermore, a comprehensive MDA design approach should allow designers to select or define suitable abstract platforms for their platform-independent designs. This leads to the formulation of a second requirement for design languages suitable for MDA: *a design language should enable the definition of appropriate levels of platform-independence.*

4 Abstract Platform Definition with MDA Standards

In this section, we pay particular attention to the definition of abstract platforms using MDA standards, namely UML 2.0 [23] and MOF 2.0 [19]. We discuss the fulfilment of the design language requirements presented in Section 3, with both the implicit and explicit abstract platform definition approaches.

4.1 Implicit Abstract Platform Definition

The concepts that plain UML prescribes for specifying communication between application parts (objects or components) imply an abstract platform that is based on request-response invocations and on message passing. In the UML 2.0 meta-model, *BehavioredClassifiers* may offer *operations* and *receptions*. *Operations* represent the capability of a classifier to receive and to respond to requests. Requests are sent when objects execute *CallOperationActions*. *Receptions* represent the capability of a classifier to receive *Signal* instances, which are sent asynchronously by other objects when these execute *SendSignalActions* and *BroadcastSignalActions*. For plain UML, the usefulness of the implicit abstract platform definition approach is restricted to abstract platforms based on request-response invocations and on point-to-point message passing.

UML has been developed as a general purpose language that is expected to be customized for a wide variety of domains, platforms and methods [25]. A certain degree of customization may be obtained in UML through semantic variation points and profiles. This choice in the definition of UML has two implications for implicit abstract platform definition: the UML specification ("plain" UML) is not conclusive with respect to the abstract platform implied, and, the customization mechanisms have to be applied in order to precisely define specific abstract platforms.

Semantic variation points provide an intentional degree of freedom for the interpretation of the UML's metamodel semantics. Some semantic variation points defined in the UML specification should be resolved for plain UML to be conclusive with respect to the abstract platform implied by the language. An example of such a semantic variation point is described in the UML 2.0 specification [23] (page 381): "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." Without resolving this semantic variation point, a designer would be forced to assume worst-case interpretations, e.g., that the implied abstract platform provides an unreliable request/response mechanism. If this is undesirable, e.g., because the abstract platform should provide a reliable request/response mechanism, a designer should resolve the semantic variation point, by defining that requests and response signals are transported reliably. Semantic variation points may be partially resolved, i.e., only for the relevant aspects. For example, a designer may consider the reliability characteristics of requests relevant, but may consider the timing characteristics irrelevant. In this case, any interpretation of the timing characteristics of requests would be acceptable. One could resolve these semantic variation points by relating the UML metamodel with a formal semantics, or to a basic set of design concepts with a formal semantics.

The specialization of UML for defining abstract platform characteristics can be made more manageable and clearly defined through the use of UML profiles. Profiles are language extensions consisting of metamodel elements that specialise elements of a reference metamodel. The specialized elements can be given specific semantics, in this way resolving semantic variation points. Furthermore, constraints expressed in a language like OCL [22] can be added to profiles to restrict the use of specific concepts or combinations of concepts. This use of profiling for implicit abstract platform definition is restricted to constraining or specialising the abstract platform implicitly defined by plain UML. In this approach, the referenced metamodel (UML 2.0's metamodel) in combination with the UML profile assumes the role of abstract platform model.

In case the relevant abstract platform characteristics cannot be represented by resolving semantic variation points through the definition of profiles, one should define new languages in terms of MOF metamodels. The design concepts of these languages are not constrained by UML, and can be arbitrarily defined through mappings from the metamodel elements to any suitable semantic domain. In this approach, the MOF metamodel assumes the role of abstract platform model. Profiling is more suited to the abstract platforms that require concepts that can be represented as specialisations of UML concepts. MOF metamodelling is suited in case the

required concepts differ too much from the UML concepts, so that a new independent metamodel has to be defined. When used systematically, profiling has the advantage that UML tools can be used for model validation and verification, since the resulting models still comply with the UML rules and constraints. MOF metamodelling has a potential drawback that available validation and verification tools may be impossible to reuse, so that new tools may have to be built for the new metamodel.

4.2 Explicit Abstract Platform Definition

As an alternative to changing the design concepts of plain UML by means of profiling and thereby changing the implicit abstract platform, we can define the abstract platform explicitly. The abstract platform is then composed with the design of the application. This can be accommodated in UML 2.0 by using model library packages [23] to define the abstract platform model. Model library packages are packages stereotyped with the standard <<*modelLibrary*>> stereotype. The abstract platform model library package can be imported by the PIM of the application. This is represented by creating a dependency between the package where the PIM is defined and the model library package where the abstract platform is defined.

An abstract platform can have an arbitrarily complex behaviour and structure, varying from a simple one-way message passing mechanism to a communication system that maintains transactional integrity and time order of messages. To make the design of complex abstract platforms manageable, we can use UML 2.0's composite structures to break up a complex design into smaller pieces. State-machine and activity diagrams may be associated with encapsulated classifiers to define their behaviour.

Since the behaviour of the abstract platform is also described in UML, it may be necessary to combine the explicit and the implicit abstract platform definition approaches, e.g., by resolving semantic variation points that are relevant for the composition of the abstract platform (explicitly defined) and the platform-independent model of the application.

5 Examples

In order to illustrate both approaches to abstract platform definition in UML, we specify the platform-independent model of a simple chatting application. This application allows users residing in different hosts to exchange text messages.

Initially, the application is described in terms of an abstract platform that supports the interaction of objects through a conference binding object. We call this abstract platform the *ConferenceAbstractPlatform*. In order to define the composition of the conference binding object with the application, we use reliable exchange of asynchronous signals. For this purpose, we define an abstract platform that supports reliable signal exchange with the implicit approach, by defining a UML profile. Later, we consider two possible realizations of the *ConferenceAbstractPlatform*, one of these relies on an event-based platform we define explicitly, and the other relies solely on the exchange of reliable signals. The relations between the different models

are depicted in Figure 2 (the *EventAbstractPlatform* is only necessary for the realization presented in section 5.4).

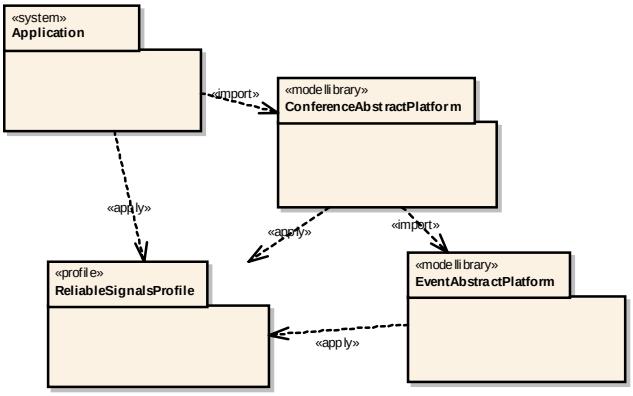

Fig. 2. Relations between the PIM of the application and the abstract platforms defined with the implicit and explicit approaches

5.1 Reliable Signal Exchange

Figure 3 depicts the *ReliableSignalsProfile* that specializes the exchange of asynchronous messages in UML 2.0. A stereotype <<*reliable*>> is defined that can be applied to instances of *SendSignalAction* (defined in the package *IntermediateActions* of the UML 2.0 meta-model). Signals created by executing a *SendSignalAction* with this stereotype are exchanged reliably, in that they cannot be lost or duplicated. The *SendSignalAction* meta-class is the only meta-class specialized in the profile. It is not necessary to specialise the meta-classes *Signal* and *Reception*, since these represent respectively, the type of signal instances exchanged and the ability to receive signal instances. The semantics of these meta-classes are independent of the manner of transmitting signal instances.

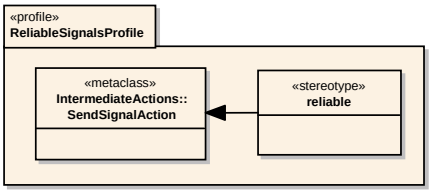

Fig. 3. A UML profile specializing the exchange of asynchronous messages

5.2 The *ConferenceAbstractPlatform*

The *ConferenceBinding* component provides the *ConferenceInterface* and requires the *ParticipantInterface*. An application part that uses the *ConferenceBinding* should provide the *ParticipantInterface*. The signals exchanged between application parts and the abstract platform are defined explicitly. A class diagram showing the

ConferenceAbstractPlatform's component, signals and interfaces is depicted in Figure 4.

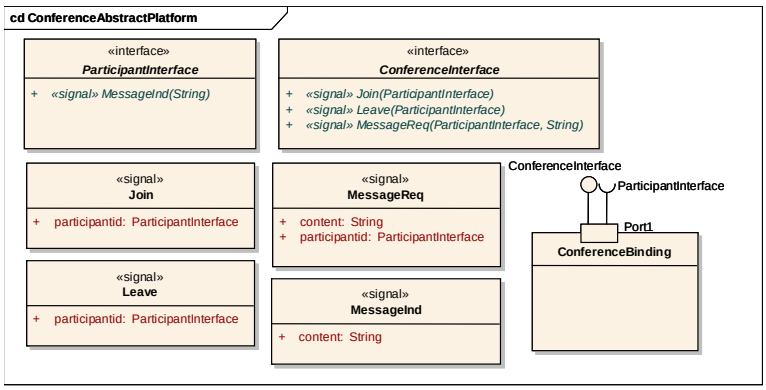

Fig. 4. The *ConferenceAbstractPlatform*

Figure 5 shows the behaviour of the *ConferenceBinding* component specified as a state-machine. *ComponentBinding* keeps a list of conference participants, which is updated whenever a *Join* or *Leave* signal is handled. Upon reception of a *MessageReq* signal, the *ConferenceBinding* sends out *MessageInd* signals to all participants of the conference. In order to simplify the behaviour we have assumed that the *MessageInd* signals are sent sequentially based on the order imposed by the list of participants (result of *i.next()*). This illustrates the use of the <<*reliable*>> stereotype.

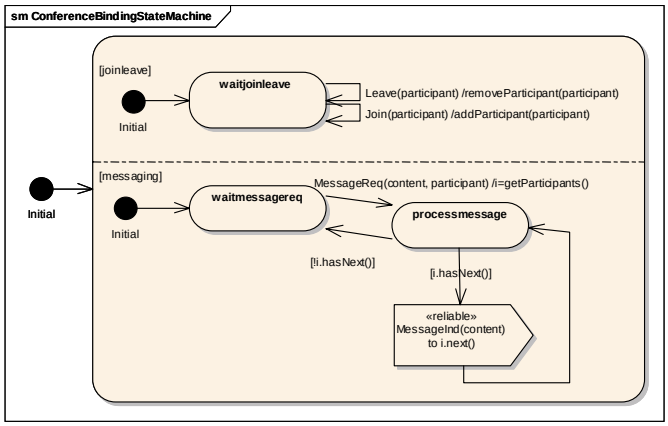

Fig. 5. The *ConferenceBinding* state-machine

The application that uses the *ConferenceAbstractPlatform* may be defined at a high-level of platform-independence, communicating with the conference binding through signal exchange. Many alternative implementations for signal exchange are possible, depending on the target platform. Further, there is a large freedom of implementation for the conference abstract platform itself. Since the application is shielded from the

internal design of the conference abstract platform, it does not depend on the interaction support eventually used by the conference binding object.

5.3 Realization of the ConferenceAbstractPlatform

Figure 6 depicts a realization of the *ConferenceBinding*. This realization relies on the abstract platform that provides reliable signals.

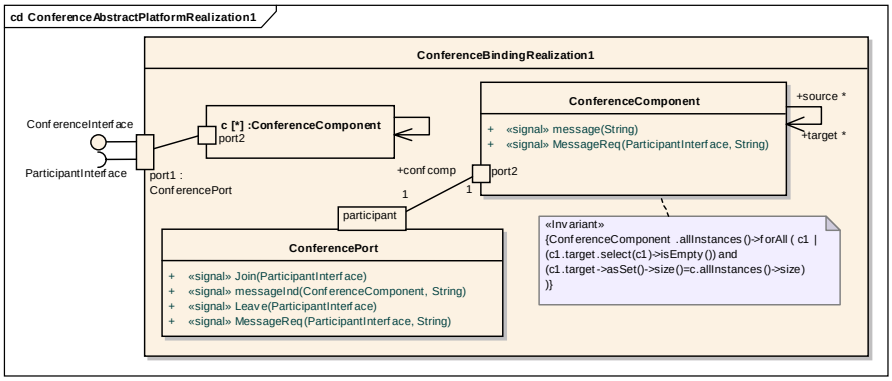

Fig. 6. A realization of the *ConferenceAbstractPlatform*

The interaction point that corresponds to *port1* is of type *ConferencePort*. The *ConferencePort* handles the signals *Join* and *Leave* and delegates the handling of signals *MessageReq* to the appropriate *ConferenceComponent*. There is a *ConferenceComponent* instance for each participant in the conference. *ConferenceComponent* instances exchange *message* signals among each other and *messageInd* with the interaction point of *port1*. The definition of these signals is omitted. An OCL [22] constraint is used to define that *ConferenceComponent* instances are fully connected, and that there are no links between an instance and itself. Figure 7 shows the behaviour associated with the *ConferenceComponent.* The behaviour of *ConferencePort* is omitted due to space limitations. The signals are exchanged reliably, and therefore, the stereotype <<*reliable*>> is applied to all *SendSignalAction* instances.

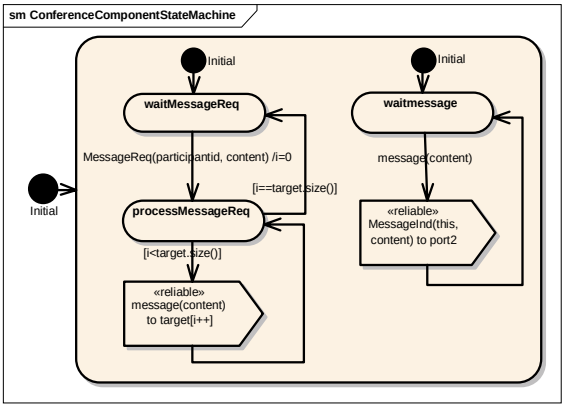

Fig. 7. Behaviour of the *ConferenceComponent* represented as a state-machine

5.4 ConferenceAbstractPlatform Realized in Terms of EventAbstractPlatform

Figure 8 depicts an alternative realization of the *ConferenceBinding*. This realization illustrates the recursive use of an explicitly defined abstract platform. The *EventAbstractPlatform* is used as part *eap* in *ConferenceBindingRealization2*. The dashed line around part *eap* is used to denote that this part is contained by reference. The multiplicity of *eap* is one, i.e., only one instance of the *EventAbstractPlatform* is used in this decomposition of the *ConferenceBinding*.

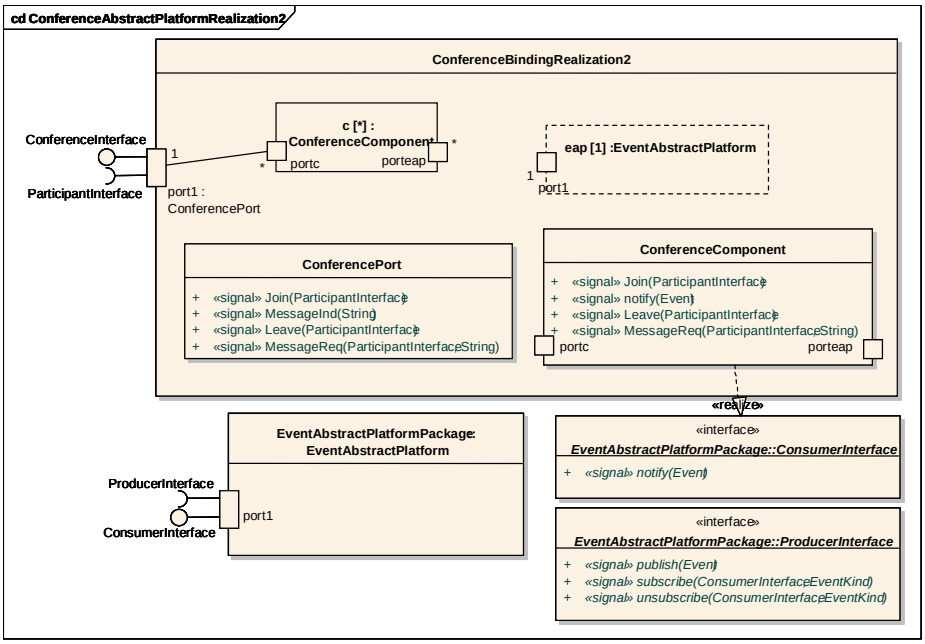

Fig. 8. Alternative realization of the *ConferenceAbstractPlatform*

The *EventAbstractPlatform* accepts events and subsequently forwards these events to objects that have subscribed to the particular event type. There is a *ConferenceComponent* for each participant in the conference. The definition of the behaviour of the *EventAbstractPlatform* is omitted here, as well as the classes *Event* and *EventKind*.

The *EventAbstractPlatform* can be realized on a number of event-based platforms, such as, e.g., JMS [27] and CORBA (with the Event Service) [16]. Alternatively, a recursive decomposition of the *EventAbstractPlatform* can be done, resulting, e.g., in a design of the *EventAbstractPlatform* that relies on a request-response abstract platform.

6 Discussion

The example from the previous section illustrates two kinds of problems that can arise when defining abstract platforms with a particular modelling language.

Firstly, a language's design concepts may force decisions about desired platform properties to be taken too early in the design process, because they do not permit abstraction of these properties. The example in the previous section illustrates this for the case of UML state machines. The state machine in Figure 5 determines that message requests are processed one at a time. Therefore, a strict interpretation of this model would exclude realizations of this abstract platform that accept multiple message requests simultaneously. Alternatively, we could have specified that a number of concurrent threads process multiple message requests at the same time. However, this alternative commits to a particular concurrency model. Ideally, we would have stated only that message requests are independent of each other, which is appropriate at the level of abstraction considered. The decision on a particular concurrency model would be delayed, and different alternative implementations would be deemed acceptable. A designer may try to mitigate the limitation of the UML representation by interpreting the behavioural specification loosely, e.g., informally defining that message requests can also be treated simultaneously despite the state machine model. However, this limits the usability of models for model transformation, automated testing, validation and simulation.

Secondly, a language's design concepts may indirectly favour some platforms over others, due to similarities in the structure of models and realizations in a particular platform. Although an implementer could try to ignore the structure and choose to adhere only to the model's semantics, he or she will be inclined to use the platform with the matching structure. The example from the previous section illustrates this for UML composite structures. In composite structures, interaction points that correspond to ports can only be created and destroyed along with the component to which they are attached. This implies that, if we want to model that an unbound number of distinct users may use the component through ports, we have to use a multiplexing scheme like the one used in Figures 6 and 8. Although the specification gives the impression that the multiplexing scheme has to be implemented, it is wiser for the implementer to ignore this scheme in case the target platform allows the dynamic creation and destruction of a component's interaction points.

7 Related Work

The MDA Guide [18] provides some examples of "generic platform types" and mentions briefly the need for a "generic platform model", which "can amount to a specification of a particular architectural style." Nevertheless, the introduction of these concepts is superficial: for example, the term "generic platform" is not even defined explicitly. In our interpretation of that documentation, we position our notion of abstract platform as subsuming that of generic platform. Abstract platforms can have other relevant characteristics in addition to defining a "particular architectural style". We have identified models that may serve as abstract platform models, in two different approaches to abstract platform definition that can be incorporated in MDA using OMG core technologies, namely UML, profiles and MOF.

The UML profile for EDOC Component Collaboration Architecture (CCA) [24] defines implicitly an abstract platform in which application part interactions are always decomposed into asynchronous messages that are exchanged through "Flow Ports". This profile also introduces the notion of recursive component collaboration (not present in UML 1.5 [26]), which can be explored to define abstract platforms explicitly, similarly to what we have obtained by using UML 2.0's composite structures.

Explicit abstract platform definition is comparable to the definition of (the behaviour of) connectors in Architecture Description Languages (ADLs), such as Rapide [11], [12] and Wright [1], when considering exclusively the characteristics of interaction support. While the role of middleware platform characteristics in ADLs have been recognized in [14], mechanisms to systematically separate and relate platform-independent and platform-specific descriptions have not been proposed in the scope of the work on Software Architecture.

8 Concluding Remarks

We have argued previously [2] that the architectural concept of abstract platform should have a prominent role in MDA development. An abstract platform defines platform characteristics that are considered at the particular level of platform-independence, and may also serve as starting point for platform-specific realization.

Design language concepts and characteristics of abstract platforms are interrelated. Therefore, careful selection of a design language is indispensable for the beneficial exploitation of the PIM/PSM separation and the definition of abstract platforms.

Often, some platform characteristics are assumed implicitly in platform-independent designs. This may lead to PIMs that cannot be reused for different platforms or it may lead to PIMs that cannot be directly compared and integrated. It may also lead to transformations that cannot be reused. Platform characteristics assumed in platform-independent designs are better understood and controlled by designers if the characteristics of the abstract platform are explicitly represented in abstract platform definitions. Furthermore, explicitly identifying an abstract platform brings attention to *balancing* between two conflicting goals: (i) platform-independent modelling, and (ii) platform-specific realization.

We have discussed how to support the concept of abstract platform in standard UML, through both the implicit and the explicit abstract platform definition approaches. In the implicit definition approach, the semantic variation points of UML should either be resolved or should be considered irrelevant for deriving intended abstract platform characteristics. UML Profiles can be useful in this approach to specialise design concepts, and manage and package abstract platforms. In the explicit definition approach, UML 2.0's composite structures are useful both for defining abstract platforms from an external and from an internal perspective. Composite structures have been a useful addition to UML 2.0. Nevertheless, we have identified some limitations with respect to the level of abstraction that can be obtained in the representation of abstract platforms with composite structures. In addition, UML 2.0 still lacks some notion of behaviour conformance in order to relate behaviours defined at a high-level of abstraction and the refined realizations of these behaviours. Consequently, we cannot formally assess the correctness of abstract platform realizations.

We have presented an example in UML in which a number of abstract platforms can be combined, both in the implicit and the explicit abstract platform definition approaches. We intend to investigate further modularisation criteria for abstract platform definitions, aiming at obtaining a reference architecture for abstract platform definition. A designer should then be able to compose an abstract platform from abstract platform definition modules. This modularisation would ideally be preserved in transformation specifications and ultimately at platform-specific level.

Acknowledgements

This work is part of the Freeband A-MUSE project. Freeband (http://www.freeband.nl) is sponsored by the Dutch government under contract BSIK 03025. This work has also been partly supported by the European Commission within the MODA-TEL IST project (http://www.modatel.org).

References

1. Allen, R. J., Garlan, D.: A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 3 (1997) 213–219
2. Almeida, J. P. A., van Sinderen, M., Ferreira Pires, L., Quartel, D.: A systematic approach to platform-independent design based on the service concept. In: Proceedings 7th IEEE Intl. Enterprise Distributed Object Computing Conference (EDOC 2003). IEEE Computer Society, Los Alamitos, CA (2003) 112–123
3. Almeida, J. P. A., van Sinderen, M., Ferreira Pires L.: The role of the RM-ODP Computational Viewpoint Concepts in the MDA approach. In: Proceedings of the 1st European Workshop on Model-Driven Architecture with Emphasis on Industrial Applications (MDA-IA 2004). CTIT Technical Report TR-CTIT-04-12. University of Twente, the Netherlands (2004) 43–51
4. Arango, G.: Domain Analysis: from Art Form to Engineering Discipline. ACM SIGSOFT Software Engineering Notes, Vol. 14, No. 3 (1989) 152–159

5. Elrad, T., Filman, R. E., Bader, A. (eds.), Communications of the ACM, Special Section on Aspect-Oriented Programming, Vol. 44, No.10 (2001) 29–97
6. Ferreira Pires, L.: Architectural Notes: a framework for distributed systems development, Ph.D. Thesis. University of Twente, Enschede, the Netherlands (1994)
7. ITU-T / ISO: Open Distributed Processing - Reference Model - Part 2: Foundations, ITU-T X.902 | ISO/IEC 10746-2 (1995)
8. ITU-T / ISO: Open Distributed Processing - Reference Model - Part 3: Architecture, ITU-T X.903 | ISO/IEC 10746-3 (1995)
9. ITU-T / ISO: Open Distributed Processing - Reference Model - Enterprise Language, ITU-T X.901 | ISO/IEC 15414:2002 (2001)
10. ITU-T: Recommendation Z.100 - CCITT Specification and Description Language. International Telecommunications Union (2002)
11. Luckham, D., Kenney, J., Augustin, L., Vera, J., Bryan, D., Mann, W.: Specification and Analysis of System Architecture Using Rapide. IEEE Transactions on Software Engineering, Vol. 21, No. 4 (1995) 336–355
12. Luckham D., Vera, J.: An Event-Based Architecture Definition Language. IEEE Transactions on Software Engineering Vol. 21, No. 9 (1995) 717–734
13. Microsoft Corporation: Microsoft .NET Remoting: A Technical Overview (2001), available at http://msdn.microsoft.com/library/en-us/dndotnet/html/hawkremoting.asp
14. Di Nitto, E., Rosenblum D.: Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. In: Proceedings of the 21st International Conference on Software Engineering (ICSE'99). Los Angeles, CA (1999)
15. Object Management Group: Model driven architecture (MDA), ormsc/01-07-01 (2001)
16. Object Management Group: Common Object Request Broker Architecture: Core Specification, Version 3.0, formal/02-12-06 (2002)
17. Object Management Group: CORBA Component Model, Version 3.0, formal/02-06-65 (2002)
18. Object Management Group: MDA-Guide, Version 1.0.1, omg/03-06-01 (2003)
19. Object Management Group: Meta Object Facility (MOF) 2.0 Core Specification, ptc/03-10-04 (2003)
20. Object Management Group: Meta Object Facility (MOF) Specification, Version 1.4, formal/02-04-03 (2002)
21. Object Management Group: MOF 2.0 Query / Views / Transformations RFP, ad/2002-04-10 (2002)
22. Object Management Group: Unified Modelling Language: Object Constraint Language Version 2.0, Draft Adopted Specification, ptc/03-08-08 (2003)
23. Object Management Group: UML 2.0 Superstructure, ptc/03-08-02 (2003)
24. Object Management Group: UML Profile for Enterprise Distributed Object Computing Specification, ptc/02-02-05 (2002)
25. Object Management Group: Unified Modelling Language (UML) Specification: Infrastructure, Version 2.0, ptc/03-09-15 (2003)
26. Object Management Group: Unified Modelling Language (UML) Specification, Version 1.5, formal/03-03-01 (2001)
27. Sun Microsystems: Java(TM) Message Service Specification Final Release 1.1 (2002), available at http://java.sun.com/products/jms/docs.html
28. World Wide Web Consortium: SOAP Version 1.2 Part 1: Messaging Framework, W3C Proposed Recommendation (2003), available at http://www.w3.org/TR/soap12-part1
29. World Wide Web Consortium: Web Services Description Language (WSDL) 1.1, W3C Note (2001), available at http://www.w3.org/TR/wsdl