On the Notion of Abstract Platform in MDA Development

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

Abstract

Although platform-independence is a central property in MDA models, the study of platform-independence has been largely overlooked in MDA. As a consequence, there is a lack of guidelines to select abstraction criteria and modelling concepts for platform-independent design. In addition, there is little methodological support to distinguish between platform-independent and platform-specific concerns, which could be detrimental to the beneficial exploitation of the PIM-PSM separation-of-concerns adopted by MDA. This paper is an attempt towards clarifying the notion of platform-independent modelling in MDA development. We argue that each level of platform-independence must be accompanied by the identification of an abstract platform. An abstract platform is determined by the platform characteristics that are relevant for applications at a certain level of platform-independence, and must be established by balancing various design goals. We present some methodological principles for abstract platform design, which forms a basis for defining 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 definition of abstract platforms and the language requirements to specify abstract platforms. We discuss how the concept of abstract platform relates to UML.

Keywords: Model-Driven Architecture (MDA), platform-independence, abstract platform, distributed application design

1. Introduction

A current trend in the development of distributed applications is to separate their platform-independent and platform-specific aspects, by describing them in separate models. 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.

A prominent development in this trend is the Model-Driven Architecture (MDA) [18, 21] development. A common pattern of MDA development is to define a platform-independent model (PIM), 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 PSMs from the same PIM, and to partially automate the model transformation process and the realization of the distributed application on specific target platforms. This may reduce development costs and improve software quality, but also forms the basis for facilitating integration, evolution and migration of software solutions, hence contributing to the limitation of maintenance costs for distributed applications.

In the context of MDA, much effort has been invested in meta-modelling (MOF [22, 23]), language definition and extension mechanisms (UML and UML profiles [26, 28]), model transformation specification (MOF Query/View/Transformation RFP [24]), and tool support. These developments constitute enabling technologies to model-driven development.

Nevertheless, the study of *platform-independence* has been particularly *overlooked*. As a consequence, there is a lack of guidelines to select abstraction criteria and modelling concepts for platform-independent design. Moreover, there is little methodological support to distinguish between platform-independent and platform-specific concerns, which could be detrimental to the beneficial exploitation of the PIM-PSM separation-of-concerns adopted by MDA.

The concept of platform-independence plays a central role in MDA development. We believe that platform-independence can only be defined once general capabilities of potential target platforms can be established. This leads to the observation that there can be PIMs 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 process, 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 approach that

accommodates designs at different levels of platform-independence.

An important architectural concept of 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 is an abstraction of infrastructure characteristics assumed for models of an application at some point of (the platform-independent phase of) the design process. 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 definition of abstract platforms and the language requirements to specify abstract platforms. We discuss how the architectural concept of abstract platform can be supported in UML 2.0 [26].

This paper is further structured as follows: Section 2 provides some background and introduces the concept of abstract platform; Section 3 provides some criteria for abstract platform definition and for the distinction between platform-independent and platform-specific concerns; Section 4 discusses how abstract platforms relate to design languages; Section 5 discusses how abstract platforms can be represented in UML; Section 6 positions our work with respect to related work. Finally, Section 7 presents our conclusions and outlines future work.

2. Platform-independent design based on the notion of 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. For the purpose of this paper, we assume that distributed applications are ultimately realized in some specific object-middleware or component-middleware technology that supports operation invocation and asynchronous message exchange, such as CORBA/CCM [19], .NET [16], and Web Services [32, 31]. Hence, a platform corresponds ultimately to some specific middleware technology.

2.1. Levels of platform-independence

When pursuing platform-independence, one could strive for PIMs that are absolutely neutral with respect to all different classes of middleware platforms. This is possible for models in which the characteristics of supporting infrastructure are irrelevant, such as, e.g., conceptual domain models [5] and RM-ODP Enterprise Viewpoint models [11] (which can be considered Computation Independent Models [21]). However, when system architecture is captured, some platform characteristics become relevant, and different sets of platform-independent modelling concepts may be used, each of which is 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. At a certain level of platform-independence, a model is said to be portable to a number of target middleware platforms. The level of platform-independence of a model must be carefully identified. We propose this identification be made an explicit step in MDA development. The notion of abstract platform, as we have proposed initially in [2], supports a designer in this step.

Figure 1 illustrates a possible hierarchy of models at different levels of platform-independence. In this figure, a highly abstract and neutral PIM is depicted at the highest level of platform-independence. Platform-independent models at a lower level of platform-independence are depicted that facilitate the transformation to two particular classes of middleware platforms, namely RPC object-based and message-oriented platforms, respectively. These latter models rely on different abstract platforms.
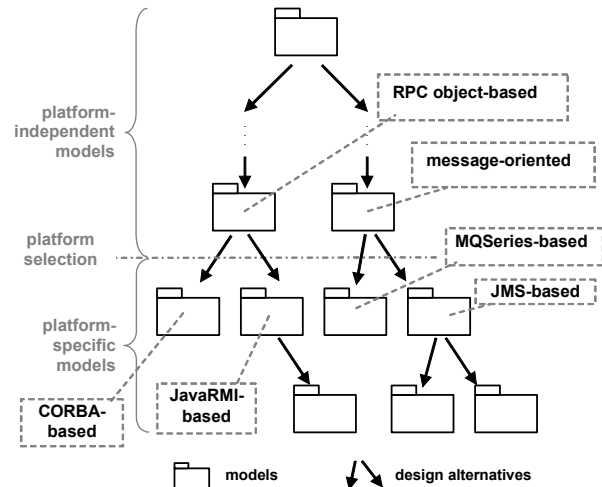

Figure 1 Models at different related levels of platform-independence

2.2. Abstract platforms

An abstract platform defines an acceptable or, to some extent, ideal platform from an application developer's point of view; it represents the support, as comprehensive and direct as possible, that is assumed by platform-independent models of a distributed application. Alternatively, an abstract platform defines characteristics that must be mappable onto the set of concrete platforms that are considered as potential targets in a development project.

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 use of the abstract platform concept may be reflected in an abstract platform model, as depicted in the in Figure 2. 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.
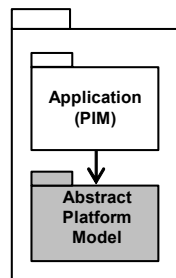

Figure 2 PIM depends on abstract platform model

2.3. Platform-specific realization

The PIM-PSM transformation is straightforward when the selected concrete platform corresponds (directly) to the abstract platform. When this is not the case, more effort has to be invested in platform-specific realization. In general, we distinguish two contrasting extreme approaches to proceeding with platform-specific realization:

1. *Adjust the concrete platform*, so that it corresponds directly to the abstract platform.

2. *Adjust the platform-specific model of the application,* while preserving the requirements specified at platform-independent level, so that the application model can be composed with the target platform model.

In approach 1, the boundary between abstract platform and platform-independent distributed 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 of this platform 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 [19], aspect-oriented programming and composition filters [6].

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 and/or the cost implications of these adjustments.

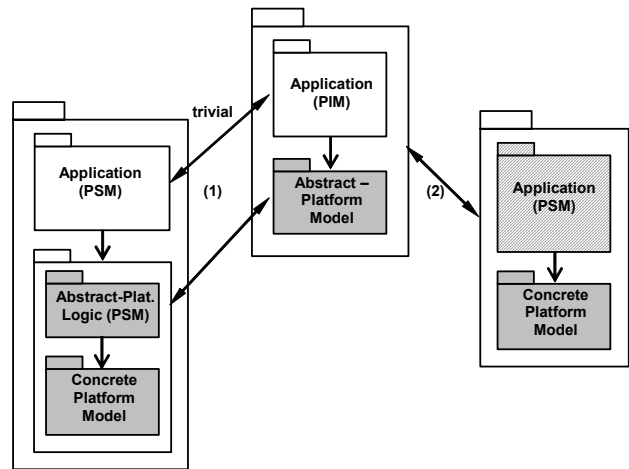Figure 3 illustrates these approaches to platform-specific realization.

Figure 3 Alternative approaches to platform-specific realization

Both approaches allow us to target different concrete platforms from the same platform-independent model. An argument against approach 1 is that it may be harder to satisfy time-performance requirements than with direct transformation (approach 2). Approach 1 may also sacrifice intuitiveness for developers that are accustomed to a particular concrete target platform. Nevertheless, this approach provides clear traceability between platform-independent and platform-specific models. Furthermore, abstract platform logic can be directly reused in the realization of other platform-independent models that rely on the same abstract platform. Approach 1 is explicitly

enabled by the identification and definition of an abstract platform, and allows us to obtain application software components that can be reused on top of different platforms [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. Abstract platform definition

The definition of an abstract platform is supported by two observations:
1. *platform characteristics may play a role in early (platform-independent) designs*, and;
2. *platform-independence must be balanced against platform-specific realization*

The first observation leads us to the conclusion that platform characteristics that play a role in platform-independent designs should be reflected in the abstract platform.

The second observation recognizes that achieving platform-independence is a requirement that must be considered in a larger context, where other relevant design goals play an important role. An MDA design process should lead efficiently to a (platform-specific) application running on a concrete platform.

The next subsections examine these observations and their implications, leading to guidelines for abstract platform design.

3.1. Role of platform characteristics

Defining an abstract platform requires the ability to identify what abstract platform characteristics are relevant at a platform-independent level. Some platform characteristics become relevant when identifying application parts and their interactions. This is the case for the characteristics of the support for interactions between system parts. Some other platform characteristics play a more subtle, but not necessarily negligible, role. Platform characteristics that may have impact in early stages of the definition of a distributed application's architecture are likely to qualify as abstract platform characteristics.

This is best illustrated by an example, in which the design of a groupware service is considered. This service facilitates the interaction of users residing in different hosts. Initially, the service designer describes the groupware service solely from its external perspective, possibly stating quality-of-service requirements on the service, e.g., that the service should have high availability. At subsequent stages of development, the designer is confronted with design decisions. In this example, we consider the following alternatives: (i) a centralized (server-based) design, and (ii) a distributed (peer-to-peer) design.

Figure 4 depicts these two solutions. In solution (i), a server facilitates the interaction between users. In solution (ii), symmetric components facilitate the interaction without the support of a centralized application-level component.

In order to improve the reusability of platform-independent models, stable aspects of a system's architecture should be captured in platform-independent models. Therefore, it would be desirable to select between alternative models (i) and (ii) during platform-independent modelling. Nevertheless, some platform-specific aspects play an important role in the selection of an adequate architecture. For example, solution (i) would introduce a single point of failure in the architecture, unless the platform provides support for replication transparency (as defined in the Reference Model for Open Distributed Processing (RM-ODP) standards [9, 10]).

Apparently, this places the designer in a dilemma, since platform selection would affect platform-independent design. In order to solve this, a designer
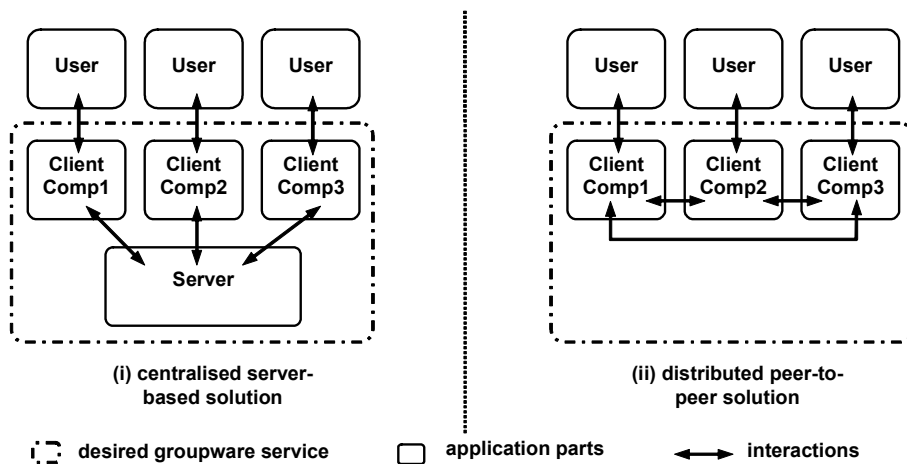

Figure 4 Alternative designs for the groupware service

should be able to express, at a platform-independent level, requirements on platform-specific realizations that would allow all design decisions that are relevant for platform-independent modelling to be captured. In our groupware service example, this would mean that requirements on the reliability of individual components should be stated at the platform-independent level, justifying the selection of a centralized or a distributed design (possibly through application of aspect-oriented modelling [8]).

Requirements expressed at a platform-independent level should justify design decisions for the design at that level and provide input for platform-specific realization. If these requirements invalidate portability requirements for platform-independent designs, then it is impossible to consider the design at the current level of platform-independence. In this case, we envision two different contrasting solutions:

(a) to consider the design at a higher level of abstraction, at which the platform characteristics are no longer relevant for design decisions taken at that level; or,

(b) to relax portability requirements, lowering the degree of platform-independence for the design. This solution reflects on the characteristics of the abstract platform being defined.

For our groupware service example, possible applications of these solutions would be:

(a) to describe the groupware service solely from its external perspective. At this level of abstraction, the reliability characteristics of the supporting infrastructure are irrelevant. Details on the service's internal design are only addressed at platform-specific modelling, and hence cannot be re-used for different target platforms; and,

(b) to restrict the set of potential target platforms, e.g., to include only platforms that provide support for highly available components. In this case, it is possible to describe the groupware service's internal design at the newly defined level of platform-independence, while still guaranteeing the satisfaction of the service requirements. The abstract platform considered provides support for highly available components.

In [3], we have presented thoroughly an example of solution (b), where an abstract platform that supports dynamic reconfiguration of components is used at some point of the design process in order to satisfy availability requirements.

3.2. Platform-independence must be balanced with platform-specific realization

Defining an abstract platform brings attention to *balancing* between two conflicting goals: (i) platform-independent modelling, and (ii) platform-specific

realization. On the one hand, an abstract platform indicates directly the support available for designers during platform-independent modelling, and therefore, reflects the needs of application designers, including the needs to handle complexity in application design and portability requirements. On the other hand, an abstract platform is established by considering the set of potential target platforms and their (common and diverging) characteristics; this bottom-up knowledge is useful to reduce the design space to be explored for platform-specific realization. Large design spaces are less amenable to automatic exploration, and require more intervention of designer, e.g., through extensive parameterization of transformations. Reducing the design space contributes to increasing the efficiency of the design process.

Poorly defined abstract platforms may lead to: applications that do not satisfy functional and non-functional requirements; platform-independent models that cannot be mapped into relevant target platforms or that that cannot resist platform evolution; platform-independent models that are too abstract, becoming less-valuable from the perspective of reuse; or complex, less reusable transformations.

The following factors should be observed when defining an abstract platform [2]:

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

These factors often depend on application domains and on specific application requirements, possibly resulting in different abstract platforms. A comprehensive MDA design approach should, therefore, allow a designer to select or define suitable abstract platforms for their platform-independent designs.

4. Abstract Platform and Design Languages

Designs must be supported by suitable design concepts and represented using suitable design languages. In an MDA development project, several design languages may be used, e.g., to produce models at different levels of abstraction. Alternatively, a single "broad spectrum" design language [7] 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 the *implicit abstract platform definition* approach, 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 [12] 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 or patterns can serve as a means to select subsets of a language's design concepts. This approach is illustrated schematically in Figure 5, where concepts are represented as geometric forms.
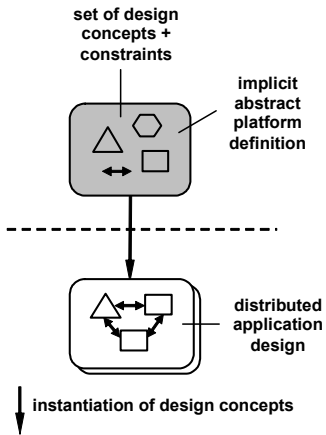

Figure 5 Abstract platform defined implicitly, by choice of design concepts

Other examples of sets of design concepts that can be used for platform-independent modelling, and that may imply the characteristics of an abstract platform, are the concepts that constitute the RM-ODP computational viewpoint such as "object", "interfaces", "operations", "streams" and "distribution transparencies" [10]. The role of computational viewpoint concepts in our MDA design approach has been discussed in [4].

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, while UML 2.0 does not support group communication as a primitive design concept, it is possible to specify the behaviour of a group communication sub-system in UML. This sub-system is then re-used in the design of the distributed application. Other examples of pre-defined artefacts that may be included in abstract platforms are the ODP trader [10] and the OMG pervasive services (yet to be defined [21]). 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. This approach is illustrated schematically in Figure 6.
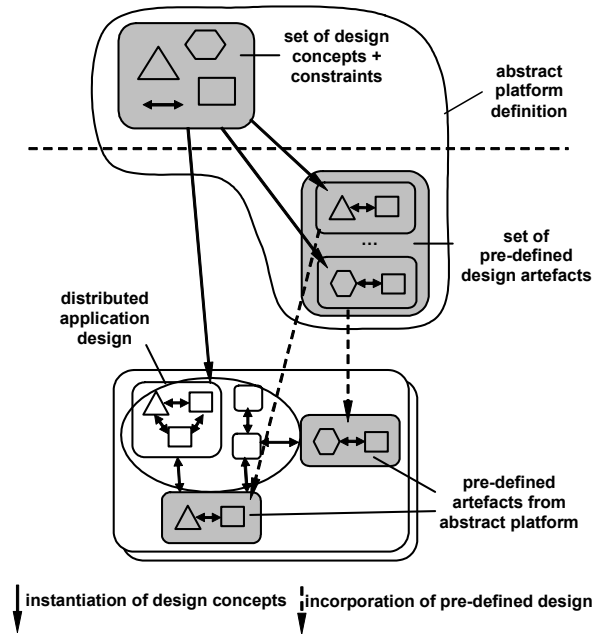

Figure 6 Abstract platform defined by incorporation of pre-defined design artefacts

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 language's design concepts should be defined precisely, so that the characteristics of the abstract platform can be derived unambiguously.* 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 allow for appropriate levels of platform-independence to be defined*.

5. 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 [26] and MOF 2.0 [28]. We discuss how to satisfy the design language requirements presented in Section 4, with the implicit and explicit abstract platform definition approaches.

5.1. 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 point-to-point message passing. Figure 7 illustrates this. Although the state-machines that describe the behaviour of the client and the server can be arbitrarily complex, the basic mechanisms for communication between the state machines are always request-response and message passing. UML assumes the existence of an implicit abstract platform between the state-machines that supports this communication mechanism. Therefore, 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.
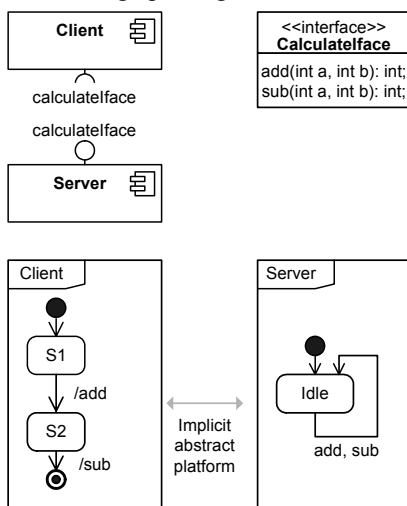

Figure 7 Abstract platform implied by UML

UML has been developed as a general purpose language that is expected to be customized for a wide variety of domains, platforms and methods [28]. A high 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 explored to 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 (page 381) [26]: "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 (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. Possibly, semantic variation points should be resolved by relating the UML metamodel with a formal semantics, or a basic set of design concepts with a formal semantics. Examples of efforts towards a formal semantics for UML are [13] and [30].

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 (e.g., in OCL [25]) 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 defined implicitly by plain UML. In this approach, the referenced metamodel (UML 2.0's
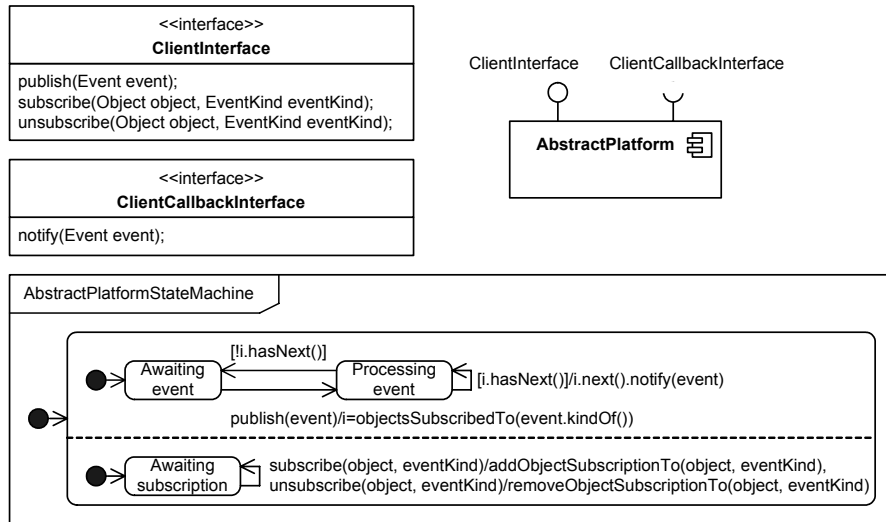
<<interface>>
ClientInterface

publish(Event event);
subscribe(Object object, EventKind eventKind);
unsubscribe(Object object, EventKind eventKind);

<<interface>>
ClientCallbackInterface

notify(Event event);

ClientInterface   ClientCallbackInterface

AbstractPlatform

AbstractPlatformStateMachine

[!i.hasNext()]

Awaiting event → Processing event → [i.hasNext()]/i.next().notify(event)

publish(event)/i=objectsSubscribedTo(event.kindOf())

Awaiting subscription ← subscribe(object, eventKind)/addObjectSubscriptionTo(object, eventKind),
unsubscribe(object, eventKind)/removeObjectSubscriptionTo(object, eventKind)

Figure 8 An explicit abstract platform definition in UML

metamodel) in combination with the UML profile assumes the role of abstract platform model.

In case relevant abstract platform characteristics cannot be represented by the capabilities offered by profiles (and semantic variation points), new languages should be defined in terms of MOF metamodels. The design concepts of these languages are not constrained by UML, and can be defined arbitrarily through mappings from the metamodel elements to any suitable semantic domain. In this approach, the MOF metamodel assumes the role of abstract platform model.

5.2. 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 included as a part of the design. This can be accommodated in UML 2.0 by using model library packages [26] (packages stereotyped as <<modelLibrary>>) as abstract platform model.

As an example, we can consider an event-based abstract platform. This abstract platform accepts signals from any object in the design and subsequently forwards these signals to objects that are willing to accept them. We can design the abstract platform by introducing an abstract platform model, which consists of an abstract platform object between communicating objects. This object must be associated with a behaviour that prescribes how objects communicate. We can specify the behaviour of the abstract platform with a state machine. In Figure 8, we use state machines to represent the behaviour of the abstract platform. Since the behaviour of the abstract platform is also described in UML in this approach, some of the remarks that were made for the implicit abstract platform definition are also valid here, particularly with respect to resolving semantic variation points.

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.
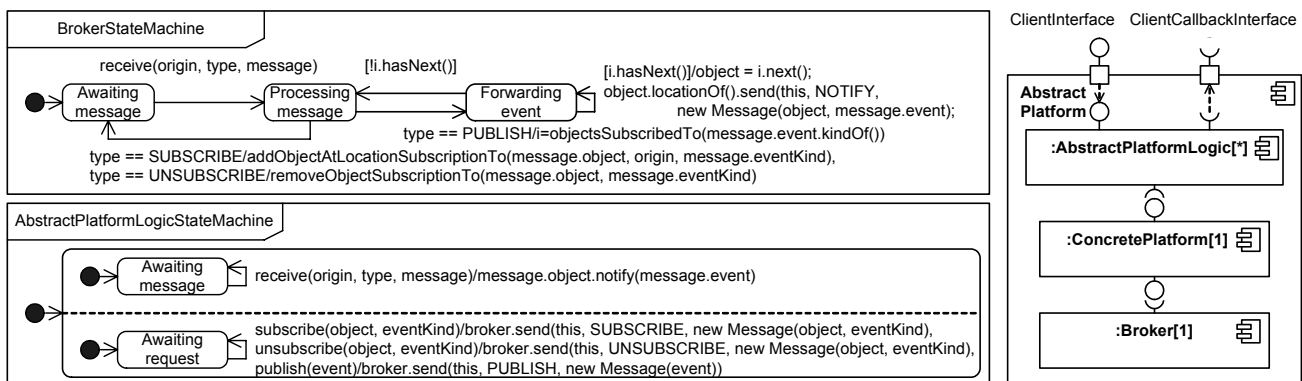
We can also use composite structures to bridge the gap

BrokerStateMachine

receive(origin, type, message)   [!i.hasNext()]

Awaiting message → Processing message ← Forwarding event ← [i.hasNext()]/object = i.next();
object.locationOf().send(this, NOTIFY, new Message(object, message.event));

type == PUBLISH/i=objectsSubscribedTo(message.event.kindOf())

type == SUBSCRIBE/addObjectAtLocationSubscriptionTo(message.object, origin, message.eventKind),
type == UNSUBSCRIBE/removeObjectSubscriptionTo(message.object, message.eventKind)

AbstractPlatformLogicStateMachine

Awaiting message ← receive(origin, type, message)/message.object.notify(message.event)

Awaiting request ← subscribe(object, eventKind)/broker.send(this, SUBSCRIBE, new Message(object, eventKind),
unsubscribe(object, eventKind)/broker.send(this, UNSUBSCRIBE, new Message(object, eventKind),
publish(event)/broker.send(this, PUBLISH, new Message(event))

ClientInterface   ClientCallbackInterface

Abstract Platform

:AbstractPlatformLogic[*]

:ConcretePlatform[1]

:Broker[1]

Figure 9 Decomposition of the abstract platform in UML

between abstract and concrete platform, using the first realization approach described in section 2.3. For example, Figure 9 shows the composite structure that bridges the gap between the abstract platform from Figure 8 and a concrete platform that only supports point-to-point message passing. The figure shows a decomposition of the abstract platform into abstract platform logic components, a broker that distributes published messages to the subscribers, and a concrete platform model. The behaviour of the concrete platform has been omitted in Figure 9 for the sake of conciseness. If an object subscribes or unsubscribes to a type of message, the abstract platform logic sends the information about the subscription to the broker, which stores this information. If an object publishes an event, the application logic sends the event to the broker. The broker then forwards the event to the abstract platform logic of each object that subscribed to the type of the event. The application logic forwards the event to the appropriate object.

The abstract platform we have presented in [3] and depicted schematically in Figure 10 is another example of this approach. This abstract platform introduces dynamic reconfiguration concepts in a platform-independent design, by specialising the notion of a component, and distinguishing between *reconfigurable* and *non-reconfigurable* components. Reconfigurable components can be *migrateable*, *replaceable* or both *migrateable* and *replaceable*. This allows a designer to establish these distinctions at a platform-independent level, specifying which components may be manipulated by operations in reconfiguration steps.
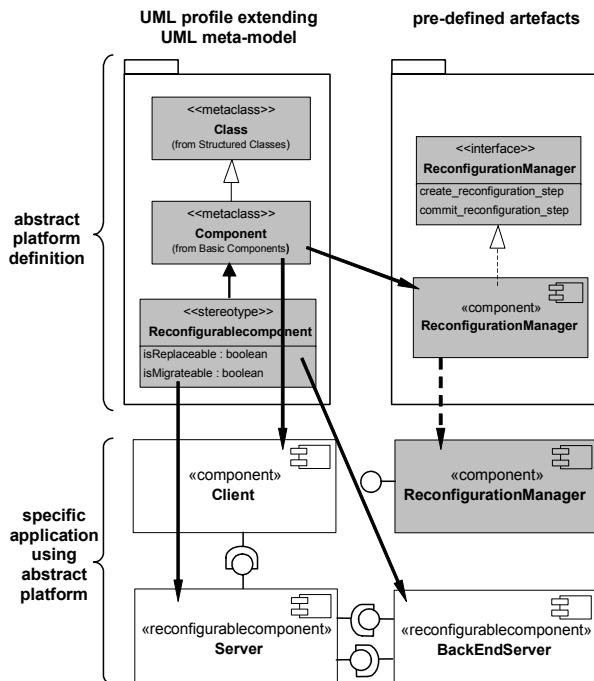

Figure 10 Support for dynamic reconfiguration in an abstract platform

In Figure 10, we represent the reconfigurable specialization of the component concept in UML 2.0 by introducing the stereotype «reconfigurablecomponent», which is applied to a UML component. This stereotype has Boolean properties isReplaceable and isMigrateable. UML statecharts can be used to specify the behaviour of (reconfigurable) components. A reconfiguration manager component represents the capabilities of the abstract platform for handling reconfiguration steps.

6.   Related Work

The MDA Guide [21] 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", as we have shown in section 3. Furthermore, we have focussed here on providing guidelines for a designer to define and represent these abstract platforms. The MDA Guide also states that a PIM "exhibits a specified degree of platform independence so as to be suitable for use with a number of different platforms of similar type." Our concept of abstract platform defines the degrees of platform independence for a PIM.

We have compared our notion of abstract platform to that of *infrastructure* as defined in the computational viewpoint of the RM-ODP [10]. This has led to a framework that allows a recursive application of the computational viewpoint at different levels of platform-independence [4].

The UML profile for EDOC Component Collaboration Architecture (CCA) [27] 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 [29]) 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 [14, 15] and Wright [1], when considering exclusively the characteristics of interaction support. While the role of middleware platform characteristics in ADLs have been recognized in [17], 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.

7.   Concluding Remarks

In this paper, we have argued that the 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.

There is no obvious distinction between platform-independent and platform-specific concerns. Therefore, a comprehensive MDA design approach should allow designers to select or define suitable abstract platforms for platform-independent designs. Explicitly identifying an abstract platform brings attention to *balancing* between two conflicting goals: (i) platform-independent modelling, and (ii) platform-specific realization.

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 abstract platform definitions are produced.

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.

We have discussed how to support the concept of abstract platform in 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.

In our future research, we will investigate modularisation criteria for abstract platform definitions. 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.

The notions of platform-independence and abstract platform should be used *judiciously*. The costs of maintaining different levels of platform-independence must not outweigh the benefits of the reuse of platform-independent models. Evaluating these costs in early stages of development is not straightforward, since the benefits of the separation PIM/PSM must be considered on the long run. This evaluation remains an open issue.

Acknowledgements

References

[1] R. J. Allen, and D. Garlan, "A Formal Basis for Architectural Connection", *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, ACM Press, July 1997, pp. 213-219.

[2] J. P. A. Almeida, M. van Sinderen, L. Ferreira Pires and D. Quartel, "A systematic approach to platform-independent design based on the service concept", *Proceedings 7th IEEE Intl. Enterprise Distributed Object Computing Conference (EDOC 2003)*, IEEE Computer Society Press,, Sept. 2003, pp. 112-123.

[3] J. P. A. Almeida, M. van Sinderen, L. Ferreira Pires and M. Wegdam, "Platform-independent Dynamic Reconfiguration of Distributed Applications", *Proceedings IEEE 10th International Workshop on Future Trends in Distributed Computing Systems (FTDCS 2004)*, IEEE Computer Society Press, May 2004, pp. 286-291.

[4] J. P. A. Almeida, M. van Sinderen, and L. Ferreira Pires, "The role of the RM-ODP Computational Viewpoint Concepts in the MDA approach", *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, ISSN 1381-3625, The Netherlands, March 2004, pp. 43-51.

[5] G. Arango, "Domain Analysis: from Art Form to Engineering Discipline", *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 3, ACM Press, May 1989, pp. 152-159.

[6] T. Elrad, R. E. Filman, and A. Bader (eds.), *Communications of the ACM, Special Section on Aspect-Oriented Programming*, vol. 44, no.10, ACM Press, Oct. 2001, pp. 29-97.

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

[8] J. Gray, T. Bapty, S. Neema, D. C. Schmidt, A. Gokhale, and B. Natarajan, "An Approach for Supporting Aspect-Oriented Domain Modeling", in *Proceedings Generative*

Programming and Component Engineering (GPCE 2003), Lecture Notes in Computer Science, vol. 2830, Springer-Verlag, Sept. 2003, pp. 151-168.

[9] ITU-T / ISO, *Open Distributed Processing - Reference Model - Part 2: Foundations*, ITU-T X.902 | ISO/IEC 10746-2, Nov. 1995.

[10] ITU-T / ISO, *Open Distributed Processing - Reference Model - Part 3: Architecture*, ITU-T X.903 | ISO/IEC 10746-3, Nov. 1995.

[11] ITU-T / ISO, *Open Distributed Processing - Reference Model - Enterprise Language*, ITU-T X.901 | ISO/IEC 15414:2002, Oct. 2001.

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

[13] J. Jürjens, A UML statecharts semantics with message-passing, in *Proceedings of the 2002 ACM Symposium on Applied Computing*, ACM Press, 2002, pp. 1009 – 1013.

[14] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and Analysis of System Architecture Using Rapide", *IEEE Transactions on Software Engineering*, vol. 21, no. 4, IEEE Computer Society Press, Apr. 1995, pp. 336-355.

[15] D. Luckham and J. Vera, "An Event-Based Architecture Definition Language", *IEEE Transactions on Software Engineering*, vol. 21, no. 9, IEEE Computer Society Press, Sept. 1995, pp. 717-734.

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

[17] E. Di Nitto and D. Rosenblum, "Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures", in *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, IEEE Computer Society Press, May 1999, pp. 13-22.

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

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

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

[21] Object Management Group, *MDA-Guide, V1.0.1*, omg/03-06-01, June 2003.

[22] Object Management Group, *Meta Object Facility (MOF) 2.0 Core Specification*, ptc/03-10-04, Oct. 2003.

[23] Object Management Group, *Meta Object Facility (MOF) Specification Version 1.4*, formal/02-04-03, April 2002.

[24] Object Management Group, *MOF 2.0 Query / Views / Transformations RFP*, ad/2002-04-10, April 2002.

[25] Object Management Group, *Unified Modelling Language: Object Constraint Language version 2.0, Draft Adopted Specification*, ptc/03-08-08, Aug. 2003.

[26] Object Management Group, *UML 2.0 Superstructure*, ptc/03-08-02, Aug. 2003.

[27] Object Management Group, *UML Profile for Enterprise Distributed Object Computing Specification*, ptc/02-02-05, Feb. 2002.

[28] Object Management Group, *Unified Modelling Language (UML) Specification: Infrastructure, Version 2.0*, ptc/03-09-15, Sept. 2003.

[29] Object Management Group, *Unified Modelling Language (UML) Specification, Version 1.5*, formal/03-03-01, March 2001.

[30] D. Varró, A Formal Semantics of UML Statecharts by Model Transition Systems, in *Proceedings ICGT 2002: International Conference on Graph Transformation,* Lecture Notes in Computer Science, vol. 2505, Springer-Verlag, 2002, pp. 378-392.

[31] World Wide Web Consortium, *SOAP Version 1.2 Part 1: Messaging Framework*, W3C Proposed Recommendation, May 2003, available at http://www.w3.org/TR/soap12-part1

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