The role of the service concept in model-driven applications development

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

Abstract

This paper identifies two paradigms that have influenced the design of distributed applications: the middleware-centred and the protocol-centred paradigm, and proposes a combined use of these two paradigms. This combined use incorporates major benefits from both paradigms: the ability to reuse middleware infrastructures and the ability to treat distributed coordination aspects as a separate object of design through the use of the service concept. A careful consideration of the service concept, and its recursive application, allows us to define an appropriate and precise notion of platform-independence that suits the needs of model-driven middleware application development.

1 Introduction

Model Driven Architecture (MDA) development is increasingly gaining support as an approach to manage system and software complexity in distributed application design [3]. MDA development focuses first on the functionality and behaviour of a distributed application, which results in a platform-independent model (PIM) of the application that abstracts from the technologies and platforms that will be used to implement it. Subsequent steps lead to a mapping from the PIM via a platform-specific model (PSM) to a platform-specific implementation (PSI). The main advantages of MDA development – software stability, software quality and return on investment – stem from the possibility to derive different PSIs (via different PSMs) from the same PIM, and to automate to some extent the model transformation process.

The concept of PIM plays a central role in MDA development. It is therefore surprising that this concept is itself ill-defined, in the sense that it is unclear which (platform-independent) properties or aspects are actually modelled and which (platform-dependent) properties or aspects are abstracted from. We believe that platform-independence can only be defined once a set of target platforms is known, such that their general capabilities and their irrelevant technological and engineering details can be established. This leads to the observation that there can be several PIMs, including various levels of PIMs, dependent on whether one wants to consider different sets of target platforms. Another observation is that different application characteristics or different sets of target platforms generally leads to different types of (intermediate) models, design structures or patterns, and model transformations.

The objective of this paper is to investigate what types of models can be useful in the MDA development trajectory, how these models are related, and which criteria should be used for their application. As a starting point for this, we analyse two basic approaches to distributed system design, viz. the protocol-centred (telecom) paradigm and the middleware-centred (distributed computing) paradigm. We consider their application to a distributed system of arbitrary complexity, consisting of several distributed application or system parts that may interact with each other and with their local user environment. A simplified model of this is depicted in Figure 1.

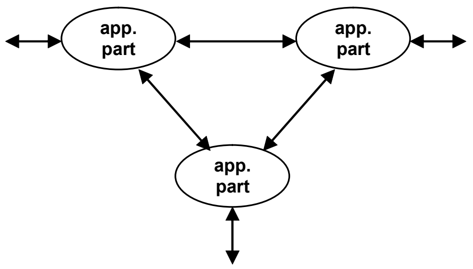

Figure 1 Model of a distributed system (application)

This paper is further structured as follows: Section 2 and Section 3 present the protocol-centred paradigm and the protocol-centred paradigm, respectively; Section 4 compares the results of the application of these paradigms on basis of a simple but effective running example; Section 5 discusses the implications of the application of the paradigms, and to this end introduces two alternative views on a distributed system; Section 6

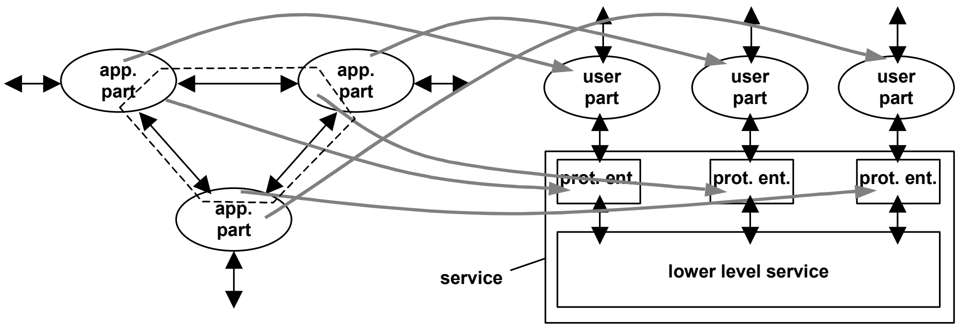

Figure 2 Model of the system in the protocol-centred paradigm

uses the results of the previous section to propose a combined use of the paradigms in a model-driven design trajectory, i.e., with models (PIMs and PSMs) associated with defined design milestones; and Section 7 presents our conclusions and outlines some future work.

2 Protocol-centred paradigm

In the protocol-centred paradigm, user parts interact locally with a *service (provider)*. A service is decomposed into *protocol entities* and a *lower level service*, which interact in order to provide the required service to user parts. The model of the system to be built consists of user parts and, for each protocol layer, a collection of protocol entities and a lower level service, as depicted in Figure 2.

The lower level service provides physical interconnection and (reliable or unreliable) data transfer between protocol entities. Lower level services can support arbitrarily complex interaction patterns, varying from connectionless data transfer (e.g., 'send and pray') to complex control facilities (e.g., handshaking with three-party negotiation).

Protocol entities communicate with each other by exchanging messages, often called Protocol Data Units (PDUs), through a lower level service. PDUs define the syntax and semantics for unambiguous understanding of the information exchanged between protocol entities. The behaviour of a protocol entity defines the service primitives between this entity and the service users, the service primitives between the protocol entity and the lower level service, and the relationships between these primitives. The protocol entities cooperate in order to provide the requested service [6].

Protocols can be defined at various layers, from the physical layer to the application layer. An application protocol defines distributed interactions that directly support the establishment of information values relevant to the application service users [7].

A systematic design method based on the protocol-centred paradigm consists of defining (i) the service to be supported in terms of the service primitives that occur at service access points, and the relationships between service primitives; and, (ii) decomposing this service in terms of a structure of protocol entities and a lower level service. This resulting structure, which we call a *protocol*, has to be a correct implementation of the service. This can be assessed formally, if both the service and protocol are specified using some formal language.

3 Middleware-centred paradigm

In the middleware-centred paradigm, system parts interact through a limited set of interaction patterns offered by a middleware platform. The model of a distributed application to be built consists of the *middleware platform* and a collection of interacting parts, often called *objects* or *components,* as depicted in Figure 3.

There are several different types of middleware platforms, each one offering different types of interaction patterns between objects or components. The middleware-centred paradigm can be further characterized according to the types of interaction patterns supported by the platform. Examples of these patterns are *request/response*, *message passing* and *message queues*.

Design methods based on the middleware-centred paradigm often consist of partitioning the application into application parts and defining the interconnection aspects by defining interfaces between parts (e.g., by using object-oriented techniques and abstracting from distribution aspects). The available constructs to build interfaces are constrained by the interaction patterns supported by the targeted platform. Examples of these

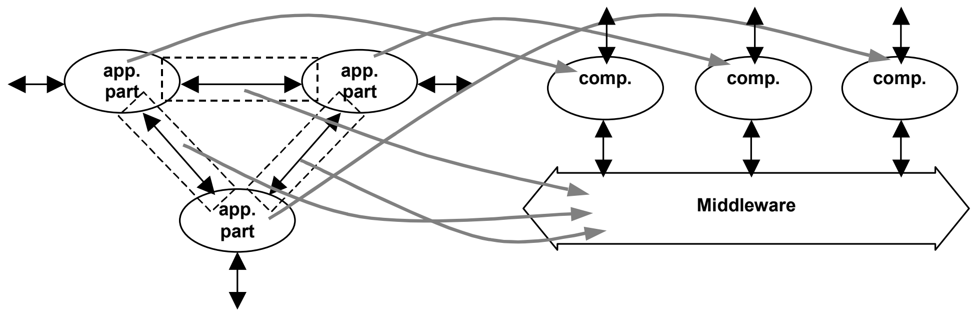

Figure 3 Model of the system in the middleware-centred paradigm

constructs are *operation invocation*, *event sources* and *sinks*, and *message queues*.

An interesting observation with respect to the middleware-centred paradigm is that it is somehow dependent on the protocol-centred paradigm: interactions between application parts are supported by the middleware, which 'transforms' the interactions into (implicit) protocols, provides generic services that are used to make the interactions distribution transparent and internally uses a network infrastructure to accomplish data transfer [8].

The middleware-centred paradigm promotes the reuse of the middleware infrastructure, facilitating the development of distributed applications. Furthermore, middleware infrastructures provide facilities to define application-level information attributes and to exchange values of these attributes through the supported interaction patterns.

4 Comparing the paradigms

When the interaction patterns between the application parts match the interaction patterns provided by the target middleware platform, a comparison of the results of the application of both paradigms is straightforward: the service provided by the application protocol corresponds to the (implicit) service provided by the middleware infrastructure. The middleware paradigm is preferred because the middleware infrastructure can be easily re-used, speeding up the development process.

When the interaction patterns are more complex, and do not match directly the interaction patterns provided by a middleware platform, the comparison requires more involvement. Therefore, we introduce our running example, the *floor-control* problem, which brings into play some complex interaction requirements.

In this example, several application parts share a set of named resources. These resources can only be used by a single application part at a time, and hence some coordination must be established in order to ensure that there is no concurrent use of a resource. Subscribers are assumed to be cooperative, i.e., they will not use the resources indefinitely. There is no pre-emption of control over a resource.

In the sequel, we present alternative solutions to the floor-control problem based on the two paradigms, and discuss the merits of both approaches.

4.1 Applying the middleware-centred paradigm

The application of the middleware-centred paradigm will lead to a number of alternative solutions, of which we consider a few. These solutions can be basically asymmetric or symmetric. In *asymmetric* solutions, an application part plays the role of a controller, centralizing the coordination of access to shared resources. Some other application parts play the role of subscribers. In *symmetric* solutions, there is no controller, and all application parts have identical roles in the coordination.

In this example, we assume a component middleware that supports remote invocation. We identify the following asymmetric solutions:

- *Callback-based*. The controller is a singleton component that has an interface with a `request_permission` operation. The parameters of this operation are the identification of the requesting subscriber and the identification of the resource. Subscribers invoke this operation to register their intention to have access to a particular resource. Eventually, when the resource is to be granted to the subscriber, a `grant` operation of the subscriber's interface is invoked by the controller.

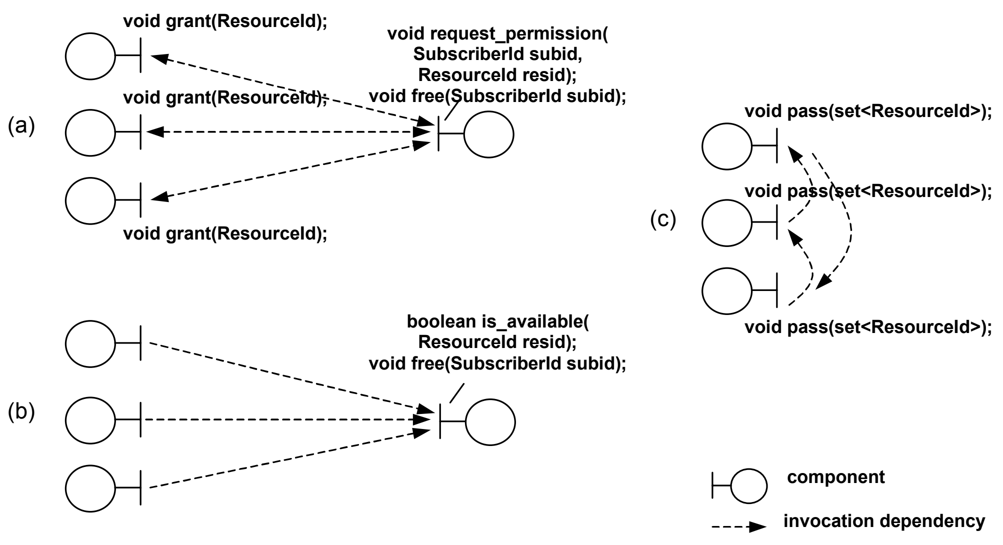
void grant(ResourceId);
void request_permission(SubscriberId subid, ResourceId resid);
void free(SubscriberId subid);
(a)
void grant(ResourceId);
void grant(ResourceId);
void pass(set<ResourceId>);
void pass(set<ResourceId>);
(c)
void pass(set<ResourceId>);
boolean is_available(ResourceId resid);
void free(SubscriberId subid);
(b)
component
invocation dependency

Figure 4 Alternative solutions for the floor-control problem in the middleware-based paradigm

When the subscriber wants to release the resource, a `free` operation of the controller's interface is invoked. This solution is illustrated in Figure 4 (a).

- *Polling-based*. The subscribers poll the controller for a certain resource by invoking the operation `is_available`, which returns the Boolean value `true` when the resource is available, and `false` otherwise. When the subscriber wants to release the resource, the operation `free` of the controller's interface is invoked. This solution is illustrated in Figure 4 (b).

We identify the following symmetric solution:

- *Token-based*. A list with the set of available resources circulates among the subscribers. Each subscriber examines the list with the set of identifiers of available resources, removes the identifier of the resource desired and forwards the list invoking an operation in the interface of the following subscriber. When a subscriber wants to release a resource, it inserts the resource identifier to be released in the list. For the sake of simplicity, we assume the set of subscribers is known a priori, so that we can ignore ring management functionality. This solution is illustrated in Figure 4 (c).

Figure 4 illustrates alternative solutions for the floor-control problem obtained by applying the middleware paradigm.

4.2 Following the protocol-centred paradigm

Following the protocol-centred paradigm, a service that interconnects application parts has to be identified: the floor-control service. We start by identifying the service primitives and their relationships. The service primitives are `request`, `granted` and `free`, with the resource identification as parameter. The identification of the subscriber is implied by the identification of the access point where the service primitive is executed.

The following relations between service primitives are informally identified:

- Local constraint: the execution of `granted` eventually follows the execution of `request` (for a given resource identification);
- Local constraint: the execution of `free` eventually follows the execution of `granted` (for a given resource identification);
- Remote constraint: a resource is only granted to one subscriber at a time.

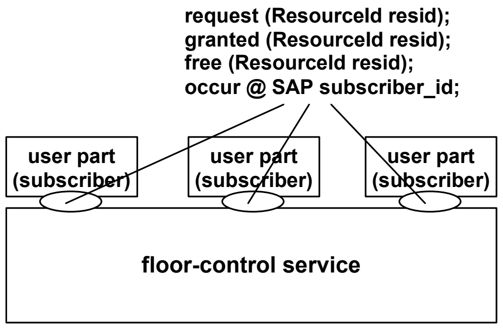

Figure 5 The floor-control service

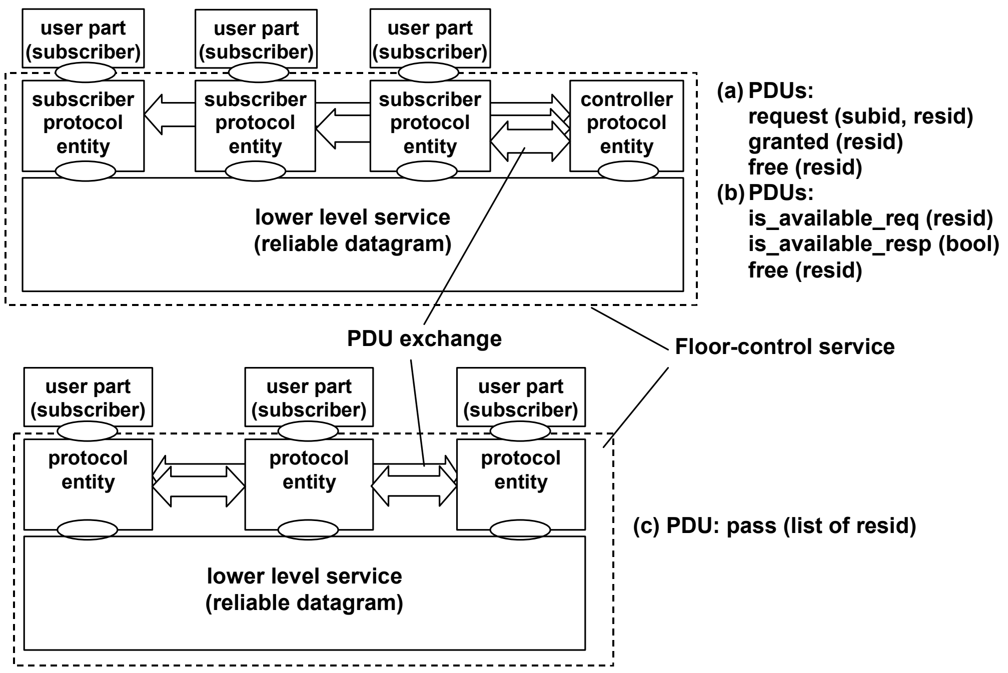

Figure 6 Alternative solutions in the protocol-centred paradigm

The service is specified in such a way that interaction requirements between application parts are satisfied without unnecessarily constraining implementation freedom. This freedom includes the structure of the service provider (the system that eventually supports the service) and other technology aspects such as operating systems and programming languages.

After the service is defined, it should be decomposed in terms of a structure of protocol entities and a lower level service. For the sake of this example, let us suppose the lower level service offers reliable transfer of a sequence of octets, which is the data transfer service used internally by middleware platforms. The protocol entities are responsible for encoding PDUs and delivering these to the lower level service.

Several alternative protocols are possible, such as:

- An asymmetric protocol similar to the *callback-based* solution, as illustrated in Figure 6 (a).
- An asymmetric protocol similar to the *polling-based* solution, as illustrated in Figure 6 (b).
- A symmetric protocol similar to the *token-based* solution, as illustrated in Figure 6 (c).

5 Discussion

Interaction patterns provided by middleware infrastructures do not always match the needs for interaction between application parts, particularly when the interactions are different than request/response patterns. In this case, interactions between application parts are often supported by an application-dependent interaction system that consists of parts of application parts and the middleware infrastructure in conjunction.

This is apparent in our floor-control example. Solutions obtained following the middleware-paradigm show that the interaction functionality is scattered across application parts, as illustrated in Figure 7.

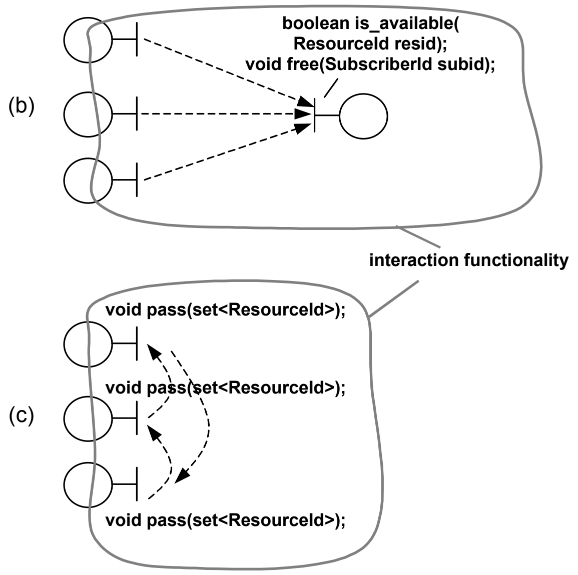
boolean is_available(ResourceId resid);
void free(SubscriberId subid);
(b)
interaction functionality
void pass(set<ResourceId>);
void pass(set<ResourceId>);
(c)
void pass(set<ResourceId>);

Figure 7 Interaction functionality is scattered across application parts

For the asymmetric solution (b), the subscriber application parts must continuously poll for a resource, in contrast with the protocol asymmetric solution (b), where the subscriber requests the resource and the *service* is responsible for "polling". In the protocol-paradigm, the service shields the application from the way in which the service is implemented. Therefore, the design of the application is not influenced by the choice of a protocol solution (the presented protocol solutions provide the same service). This is not the case for the middleware-approach, where the set of interaction patterns supported by the middleware directly influence the design of the application parts. Analogous arguments are applicable to solutions (a) and (c).

Applying the middleware paradigm for applications with complex interaction requirements, yields similar results to following the protocol approach *without* considering the required service explicitly. As has been pointed in [9], the definition of services should precede or accompany, but definitely not follow, the specification of protocols. The use of the service concept leads to careful consideration of the interaction problem being addressed. In terms of system structure, the use of the service concept promotes an appropriate application of the layering principle. For that, the principles of orthogonality (separation of concerns) and generality [7] should be observed when devising the service definition.

We distinguish two alternative views on a distributed system, namely, a view in which the interaction systems provided by the middleware platform are recognized as separate objects of design (Figure 8) and a view in which the application-dependent interaction systems between application parts are recognized as separate objects of design (Figure 9). In the former view, the design of application parts is predominant, and in the latter view, the design of the application-dependent interaction systems shifts to the foreground.

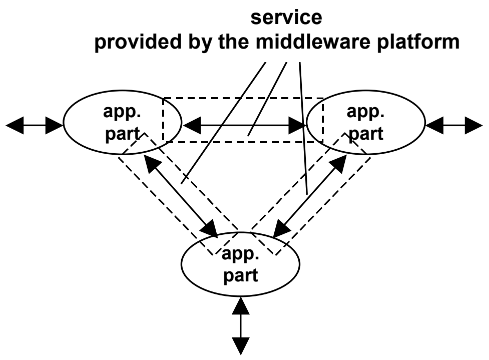

Figure 8 Focus on interaction systems provided by the middleware

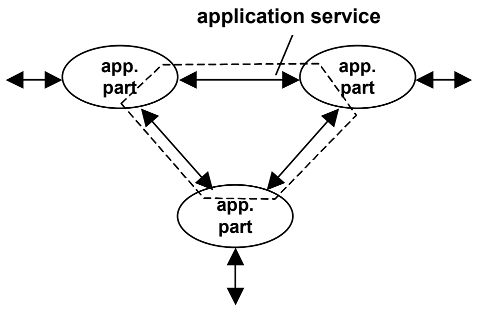

Figure 9 Focus on application-dependent interaction system

Whether or not the design of application-dependent interaction system is part of the design process depends on the application requirements and on the objectives of the designer [7]. In the following situations, interaction system design should be considered:

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

The design of the interaction system implies explicit attention to design choices that concern the effectiveness and efficiency of interactions. For example, QoS aspects that are influenced by distribution aspects are better addressed separately.

We observe that the middleware-paradigm leverages the reuse of a large building block that provides an interoperability architecture across programming languages, operating systems, network technologies and the support for application data types. We argue that interaction systems provided by the middleware are suitable for building application interaction systems.

6 Combined use of the paradigms

6.1 Platform-independence

The term *platform* is used to refer to technological and engineering details that are *irrelevant* to the fundamental functionality of a system (part). A platform-independent model is a model that does not depend on, or rely on characteristics of a particular platform. In order to refer to platform-independent or platform-specific models, one must define what a platform is, i.e., one must define which technological and engineering details are irrelevant *in a particular context*. We assume in this paper that platform corresponds to some specific middleware technology, such as, e.g., CORBA/CCM, J2EE, .NET or Web Services.

Ideally one could strive for PIMs that are absolutely neutral with respect to different classes of middleware technologies. However, we foresee that at different stages of the development trajectory, different sets of platform-independent modelling concepts may be required for different classes of target middleware platforms. Figure 10 illustrates a possible MDA design trajectory, in which such a highly abstract and neutral PIM is depicted as the starting point of the trajectory. In Figure 10, the platform-independent models are defined that facilitate the transformation to two particular classes of middleware platforms, namely RPC-based (object-based) and asynchronous messaging (message-oriented) platforms, respectively.

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

6 Milestones in the Model-driven Design Trajectory

In the combined use of the protocol-centred and middleware-centred paradigms, the following milestones are defined along the design trajectory:

- *Service definition*. The service definition sets the boundaries of the application interaction system to be designed. Services are specified at a level of abstraction in which the supporting infrastructure is not considered. A service specification focuses solely on the behaviour as observed from the user of a service. In our case, the infrastructure is the middleware platform, and therefore, service specifications are middleware-platform-independent. The service concept defines a platform-independent level that is also “paradigm”-independent (as in [1]), in the sense that a service may be implemented by a broad set of middleware platforms that support different interaction patterns (and would be, e.g., positioned at the top of the trajectory of Figure 10). Application parts that rely on the service definition may be defined on the same level of platform-independence.

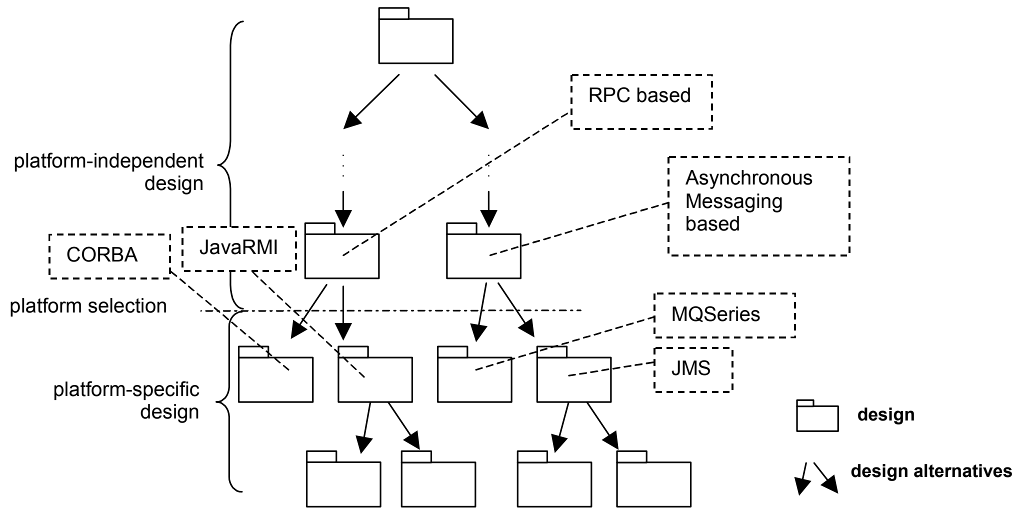

Figure 10 MDA design trajectory

- *Platform-independent service design*. The platform-independent service design consists of the *platform-independent service logic*, which is structured in terms of *service components*, and an *abstract-platform definition*. The choice of abstract platform definition must consider the portability requirements since it will define the characteristics of the platform upon which service components may rely. The level of abstraction at which the platform-independent service logic is specified depends on the abstract platform definition. Figure 11 illustrates the service definition and platform-independent service design milestones.
- *Abstract-platform realization*. The abstract platform definition is matched with a concrete platform definition. This may be straightforward when the selected platform conforms (directly) to the abstract platform definition. The abstract platform definition characterizes the level of abstraction at which platform-independent service logic is specified. Concepts used for the elaboration of platform-independent models may differ from the concepts available in target platforms, since the former concepts should be generic enough to allow a mapping to possibly different sets of the latter concepts. This difference has to be accommodated when the path to realisation is taken. For each concept represented in a platform-independent model, there should be a corresponding concept or a corresponding combination of concepts in the target platform. When this is not the case, recursion of the application of the service design step may be necessary, with the abstract-platform definition functioning as service definition for the recursion. In this recursion, the functionality of the abstract-platform is leveraged with the addition of *abstract-platform service logic*, which is a platform-specific model defined in terms of the concrete platform. Figure 12 illustrates the recursive application of the service concept.

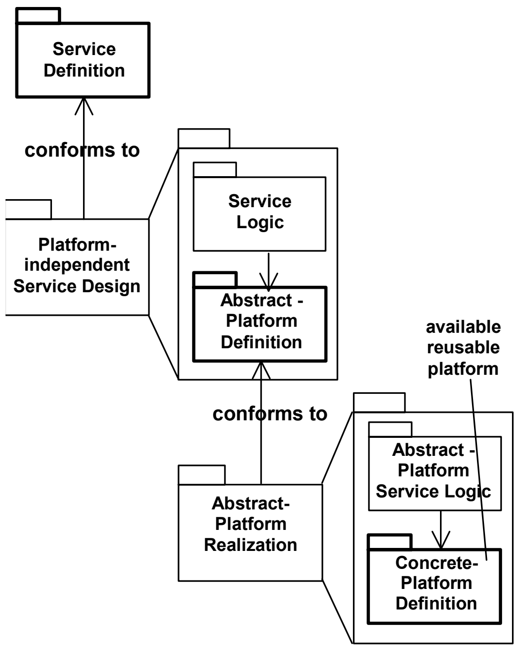
Service Definition
conforms to
Platform-independent Service Design
Service Logic
Abstract - Platform Definition
available reusable platform
conforms to
Abstract-Platform Realization
Abstract - Platform Service Logic
Concrete-Platform Definition

Figure 12 Recursive application of the service concept

Alternatively to recursive application the service, platform-specific realization may proceed with direct transformation with no preservation of the border between abstract platform and service logic. For each concept represented in a platform-independent model, there should be a corresponding concept or a corresponding combination of concepts in the target platform.

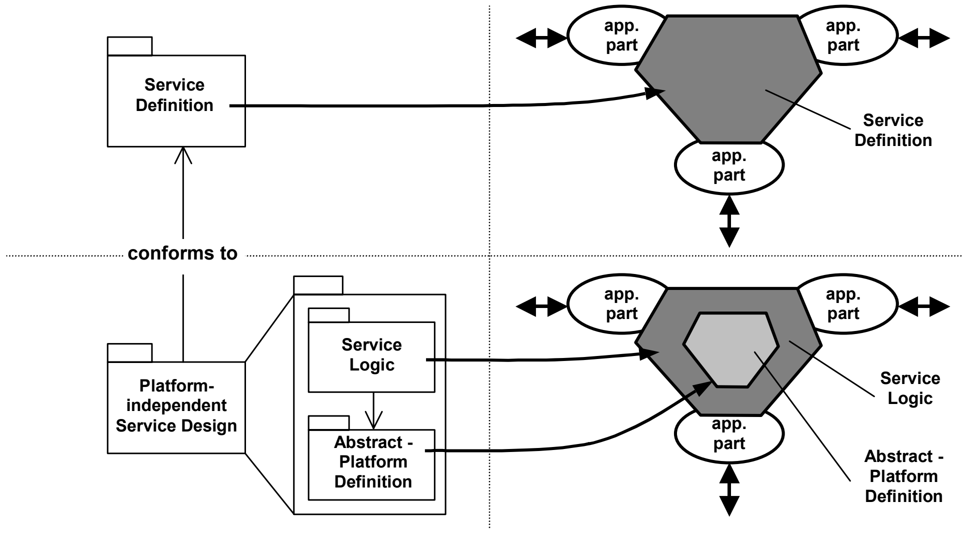

Figure 11 Milestones in the design trajectory

7 Conclusions

We have argued the case for an increased role of service specifications in the design and model-driven development of distributed applications. A careful consideration of the service concept, and its recursive application, allows us to define an appropriate and precise notion of platform-independence.

We have described two paradigms to approach the design of distributed applications: middleware-centred and protocol-centred. A combined use of both paradigms will give us the following benefits:

- reuse of middleware infrastructures;
- use of service specifications to address interaction concerns explicitly (e.g., to tackle the complexity of complex coordination problems);
- reuse of knowledge in application protocol methodologies (such as, e.g., [7]);
- an approach to target different platforms, i.e., through, possibly recursive, application of service specification and design. In this approach, service specifications provide stable reference points in the development process.

Current research focuses on elaborating the proposed model-driven development approach, and demonstrating its applicability through case studies. Furthermore, we are identifying requirements for a modelling language to support the approach. This language should facilitate the specification of services and their designs (at multiple abstraction levels), and have a formal basis to develop techniques for testing or proving the correctness of service designs.

Acknowledgements

This work is partly supported by the European Commission in context of the MODA-TEL IST project (`http://www.modatel.org/`) and the Telematica Instituut in the context of the ArCo project (`http://arco.ctit.utwente.nl/`).

References

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

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

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

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

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

[6] R. Sharp. *Principles of protocol design*. Prentice-Hall International Series in Computer Science, Prentice-Hall, Great Britain, 1994.

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

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

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