Abstract Platform and Transformations for Model-Driven Service-Oriented Development

João Paulo A. Almeida[1,2], Luís Ferreira Pires[2], Marten van Sinderen[2]

[1]Telematica Instituut, P.O. Box 589, 7500 AN Enschede, The Netherlands

[2]Centre for Telematics and Information Technology, University of Twente, P.O. Box 217, 7500AE, Enschede, The Netherlands

Abstract. In this paper, we discuss the use of abstract platforms and transformation for designing applications according to the principles of the service-oriented architecture. We illustrate our approach by discussing the use of the service discovery pattern at a platform-independent design level. We show how a trader service can be specified at a high-level of abstraction and incorporated in an abstract platform for service-oriented development. Designers can then build platform-independent models of applications by composing application parts with this abstract platform. Application parts can use the trader service to publish and discover service offers. We discuss how the abstract platform can be realized into two target platforms, namely Web Services (with UDDI) and CORBA (with the OMG trader).

1 Introduction

The Model-Driven Architecture (MDA) has been introduced as an approach to manage system and software complexity in distributed application design. MDA defines a set of basic concepts such as model, metamodel and transformation, and proposes a classification of models that offer different abstractions [16]. The main benefits of software development based on MDA – software stability, software quality and return on investment – stem from the possibility to derive implementations of an application in different platforms from the same platform-independent models (PIMs), and to automate to some extent the model transformation process.

Service-oriented computing (SOC) promises to deliver the methods and technologies to facilitate the development and maintenance of distributed (enterprise) applications [21]. The service-oriented paradigm is in essence characterized by the explicit identification and description of the externally observable behaviour, or service, of an application. Applications can then be discovered and linked, based on the description of their externally observable behaviour [22]. According to this paradigm, developers in principle do not need to have knowledge about the internal functioning and the technology-dependent implementation of the applications being linked. Often the term service-oriented architecture (SOA) is used to refer to the architectural principles that underlie the communication of applications through their services [8].

We can observe from the above that service-oriented computing and model-driven engineering share some common goals, namely they both strive to facilitate development and maintenance of distributed enterprise applications, although they achieve these goals in different ways. In this paper we discuss a combination of MDA and SOA, resulting in a model-driven service-oriented development approach that can profit from the benefits of both these developments.

In particular, this paper provides the following contributions to model-driven service-oriented development:

1. we prescribe how services can be modelled in a platform-independent manner. For that, we use a general-purpose behaviour modelling language called Interaction Systems Design Language (ISDL) [13, 23] in combination with UML [19] and OCL [18];
2. we incorporate the service discovery pattern to the platform-independent design level. Our solution consists of modelling a trader service at a high-level of abstraction, and including it in an *abstract platform for service-oriented development*. This enables designers to build platform-independent models of an application by composing application parts with this abstract platform. Application parts can then use the service trader to publish and discover service offers;
3. we discuss the implementation (via transformations) of platform-independent models into two target platforms, namely Web Services [27, 28] (with UDDI repositories [15]) and CORBA (with the OMG trader [17]). We discuss how the characteristics of the abstract platform are accommodated during this transformation step.

The paper is organised as follows. Section 2 presents an overview of the different levels of models and model transformations addressed in this paper. Section 3 presents the proposed abstract platform for service-oriented development. Section 4 discusses the implications of the abstract platform for model transformations that lead to platform-specific realisations, and illustrates the approach with an application example. Finally, Section 5 summarises our results and indicates topics for future work.

2 Design Process Overview

We consider the following organization of the model-driven service-oriented development process into different levels of models: (i) the application service specification level, which describes the services offered by application parts to their environment; (ii) the platform-independent application design level and (iii) the platform-specific application design level. In this paper, we focus on the latter two levels.

The platform-independent application design level describes services that make use of an abstract platform [3, 5]. This abstract platform consists of an abstraction of service infrastructure characteristics that are assumed for the platform-independent design level. The abstract platform we discuss here supports the service discovery pattern at a platform-independent design level, and is further referred to as *SOA* *trader abstract platform* in this paper. The service discovery pattern we adopt uses a trader, with which potential service consumers interact to find services based on service properties [26].

The platform-specific application design level describes the realisation of the platform-independent application design for a particular middleware platform. In order to show the flexibility of the relation between the platform-independent application design level and the platform-specific application design level two different middleware platforms are used, namely, Web Services and CORBA.

Fig. 1 depicts the organisation of the design trajectory we assume in this paper, with the three aforementioned levels of models. It reveals the composition of application services and the two elements that form the SOA trader abstract platform (in grey): the service trader and the underlying SOA abstract platform. In addition, it reveals the use of two target concrete platforms, namely Web Services and CORBA. Model transformations are depicted as arrows from a source model to a target model.

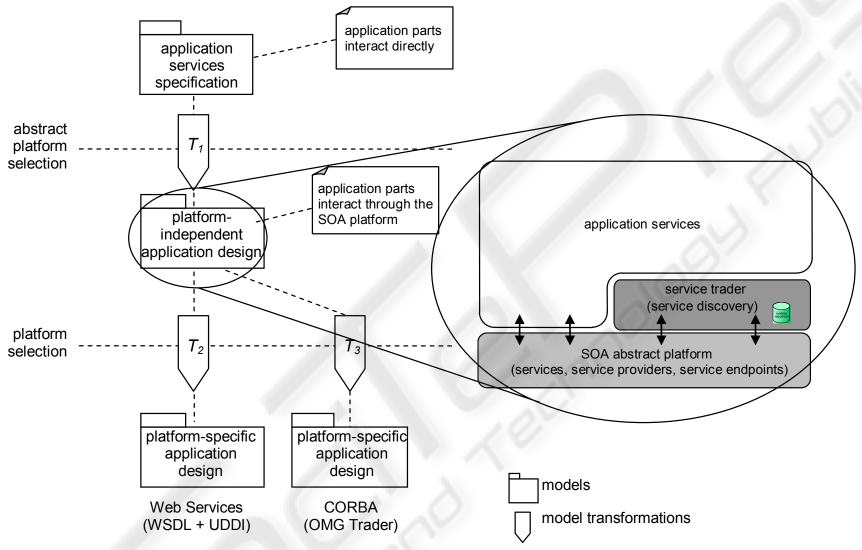

application services specification
application parts interact directly
abstract platform selection
T_1
platform-independent application design
application parts interact through the SOA platform
application services
service trader (service discovery)
SOA abstract platform (services, service providers, service endpoints)
platform selection
T_2
T_3
platform-specific application design
platform-specific application design
Web Services (WSDL + UDDI)
CORBA (OMG Trader)
models
model transformations

Fig. 1. Design trajectory consisting of three levels of models.

3 The SOA Trader Abstract Platform

This section defines the elements of the SOA abstract platform. We combine the two abstract platform definition approaches we have defined in [4]: the *language-level approach* and the *model-level approach*. In the language-level approach, the characteristics of an abstract platform are implied by the set of modelling constructs, patterns and styles used to model the application. For example, using “signals” in UML implies an abstract platform based on asynchronous messaging. In the model-level approach, the characteristics of an abstract platform are implied by the set of design artefacts that comprise the abstract platform. The trader service defined in this paper is an example of such a design artefact. An application designer can build the application by composing application parts with the abstract platform. In this approach, the modelling language is used to describe: (i) the application, (ii) any design artefacts included in the abstract platform, and (iii) the composition of the application and these artefacts.

3.1 Overview

We first define the underlying *SOA abstract platform*, using a language-level approach. The language adopted for this level is ISDL [13], which is suitable for the definition of services and their interactions. This language has a formal semantics and conformance rules, which allow one to assess the conformance of behaviour refinements. The concepts in ISDL are not constrained by UML, and provide better support for the middleware-platform-independent modelling of interactions, as argued in [2]. We use UML class diagrams to model information attributes used in ISDL behavioural specifications, and OCL to model constraints on these attributes. The ISDL metamodel is defined as a MOF metamodel in [7], which facilitates its combination with UML and OCL. Fig. 2 depicts the modelling constructs of ISDL, UML and OCL schematically (language-level).

The *SOA trader abstract platform* is built on top of the underlying service-oriented abstract platform and is defined with a model-level approach. This abstract platform provides a trader service, which is defined in ISDL. Information attributes (e.g., service offers) are described with UML. The use of a trader service is a well established pattern of service discovery in service-oriented architectures. Examples of service traders in middleware platforms are the OMG CORBA trader [17] and the UDDI registry [15] (a Web Services technology). Our trader service resembles the trading function that has been defined in the scope of the Reference Model for Open Distributed Processing (RM-ODP) [14, 11].

Fig. 2 shows schematically how the elements of the SOA trader abstract platform are defined and incorporated in the platform-independent application design.

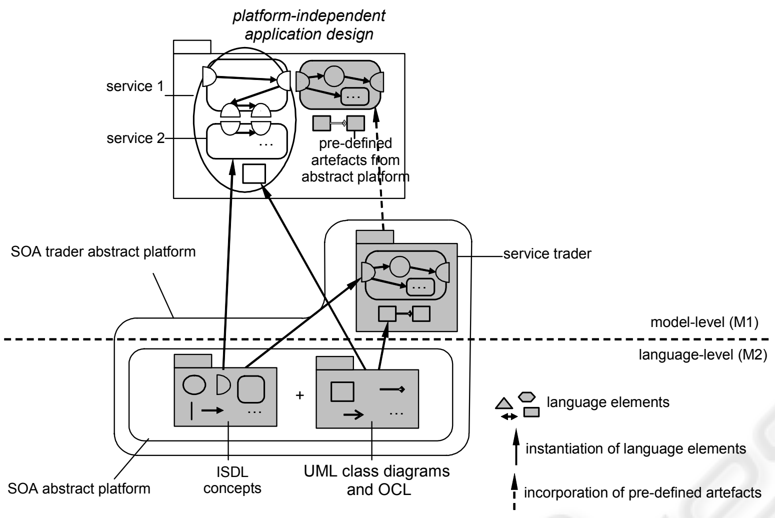

platform-independent application design
service 1
service 2
pre-defined artefacts from abstract platform
SOA trader abstract platform
service trader
model-level (M1)
language-level (M2)
language elements
instantiation of language elements
incorporation of pre-defined artefacts
SOA abstract platform
ISDL concepts
UML class diagrams and OCL

Fig. 2. SOA trader abstract platform definition and usage.

3.2 SOA Abstract Platform

The SOA abstract platform supports the interaction of various (potentially distributed) service providers through their services. The concept of abstract interaction discussed in [2, 13] is suitable for this purpose. In ISDL, behaviours are defined in terms of (abstract) actions and interaction contributions and constraints on them. Since services only concern observable behaviour, at this level behaviours only contain interaction contributions.

An abstract interaction models the successful completion of a shared activity between the interacting parts, and establishes a result at some location and some time. Constraints can be defined to restrict the results of information established in the interaction, and to restrict which behaviours are allowed to interact with each other. In general, each interacting party constrains the attributes established as result of an interaction: a party may offer a set of values, accept a set of values, or both. These constraints on values supply different ways of cooperation [24], namely, *value passing*, *value checking* and *value generation*. Value passing occurs when an interacting party offers a value and the other parties accept this value. Value checking occurs when all interacting parties offer the same value. In value generation, the interacting parties offer a range of acceptable values and the interaction happens if it is possible to establish a value that matches all requirements. The SOA abstract platform supports only value passing, since this is a more suitable abstraction of the support provided by target platforms.

Fig. 3 illustrates the ISDL notation with a simple service client/provider example. It shows an example of a structured behaviour (of name *Composition*), which consists of five behaviour instantiations (of names *c1*, *c2*, *c3*, *s1* and *s2*) of two behaviour types (of names *ClientBehaviour* and *ProviderBehaviour*). An interaction contribution is represented by a semi-circle drawn on the border of the behaviour in the context of which it is defined.

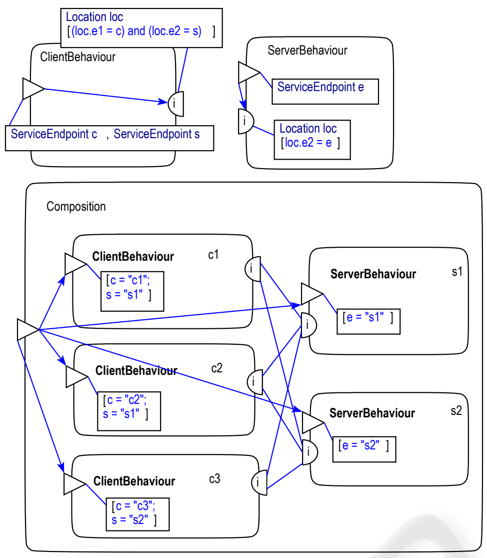


Fig. 3. Example of usage of SOA abstract platform (exported from Grizzle [9]).

In Fig. 3 each interaction is represented by two interaction contributions connected by a line. We use a composite location type (*Location*), which consists of two (interchangeable) service endpoints (*ServiceEndpoint*). A constraint of an interaction contribution is drawn on a box attached to the interaction contribution. In this example, the location constraints are such that servers may interact with any client. The clients constrain location such that *c1* only interacts with *s1*, *c2* only interacts with *s1* and *c3* only interacts with *s2*. Arrows represent enabling causality relations between interaction contributions, and triangles represent entry points that allow behaviours to be instantiated with some parameter values.

Fig. 4 shows the UML class diagram that defines the location attribute type *Location* used at the platform-independent application design level.

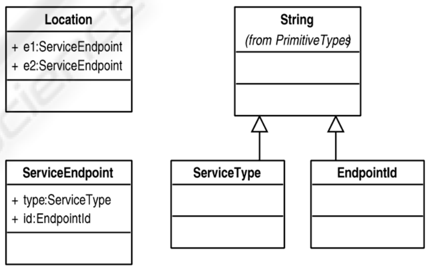


Fig. 4. Location and ServiceEndpoint classes.

3.3 SOA Trader Platform

In order to allow for service discovery, the SOA trader abstract platform contains a trader service, which registers a number of service offers. Fig. 5 depicts the classes relevant to service offers.

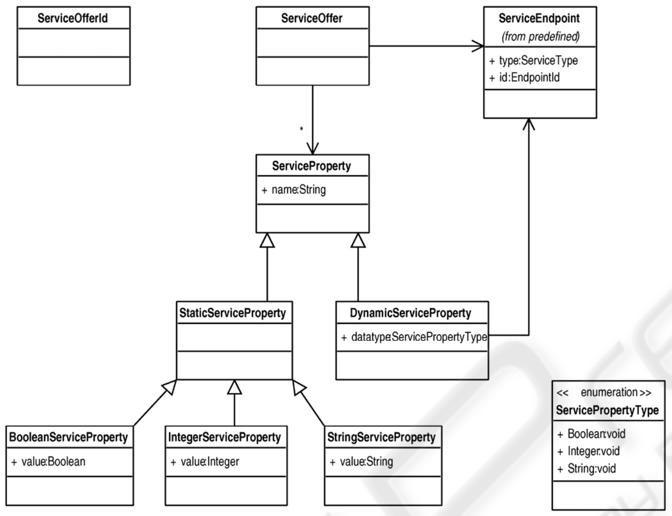


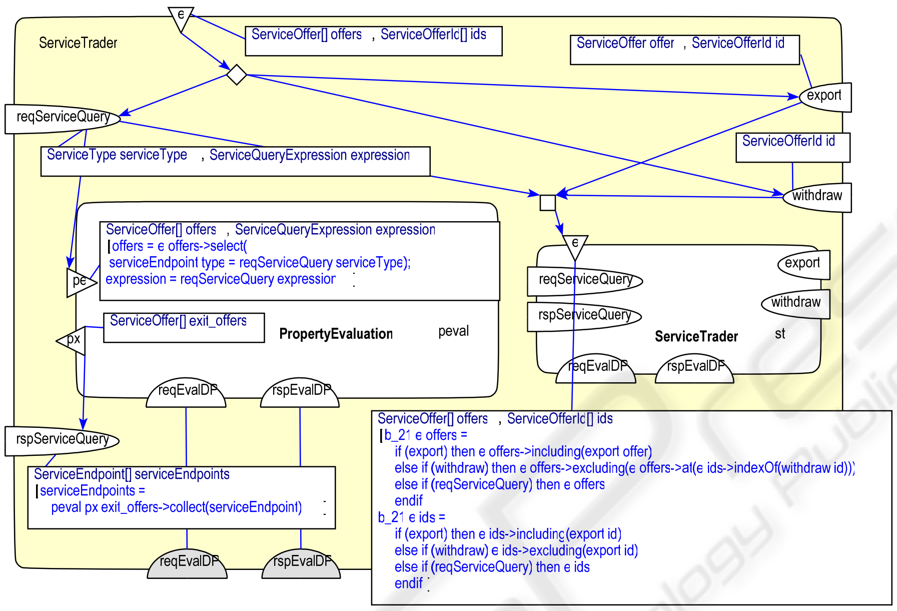
Fig. 5. Service offers.

Service offers (instances of *ServiceOffer*) are represented as information attributes, exchanged with the trader in an *export* interaction. Service offers include a service endpoint (an instance of *ServiceEndpoint*) and a number of service properties (instances of *ServiceProperty*). A service endpoint in a service offer determines how the service represented by this service offer can be accessed. An application part that accesses a service should refer to the service endpoint that corresponds to the desired service. This can be done by properly constraining the location attribute.

Service properties may be either static or dynamic. Static properties have immutable values, which are determined when a service provider exports a service offer. Dynamic properties are evaluated dynamically when a lookup operation is performed [26]. Each static service property consists of a name-value pair. In Fig. 5 these pairs are represented by the attributes of the subclasses of *ServiceProperty*. Each dynamic service property consists of a service endpoint (instance of *ServiceEndpoint*) and a service property type (value of the *datatype* attribute). The service endpoint associated to a dynamic service property is used by the trader to inspect the current value of the dynamic property. The service property type identifies the type of the dynamic property.

A client of the trader service specifies a service query by providing a service type (*ServiceType*) and an expression (*ServiceQueryExpression*) involving service properties (*ServiceProperty*). *ServiceQueryExpression* includes support for basic arithmetic and Boolean operators. The definition of *ServiceQueryExpression* is omitted here due to space restrictions (we refer to [6] for details).

Fig. 6 depicts the behaviour definition of the trader service in ISDL. A *reqServiceQuery* interaction is followed by the execution of the *PropertyEvaluation* behaviour that evaluates the service query expression. Its *exit_offers* exit parameter represents a sequence of offers that comply with the service query.



```
ServiceOffer[] offers , ServiceQueryExpression expression
|offers = e offers->select(
 serviceEndpoint type = reqServiceQuery serviceType);
expression = reqServiceQuery expression
```

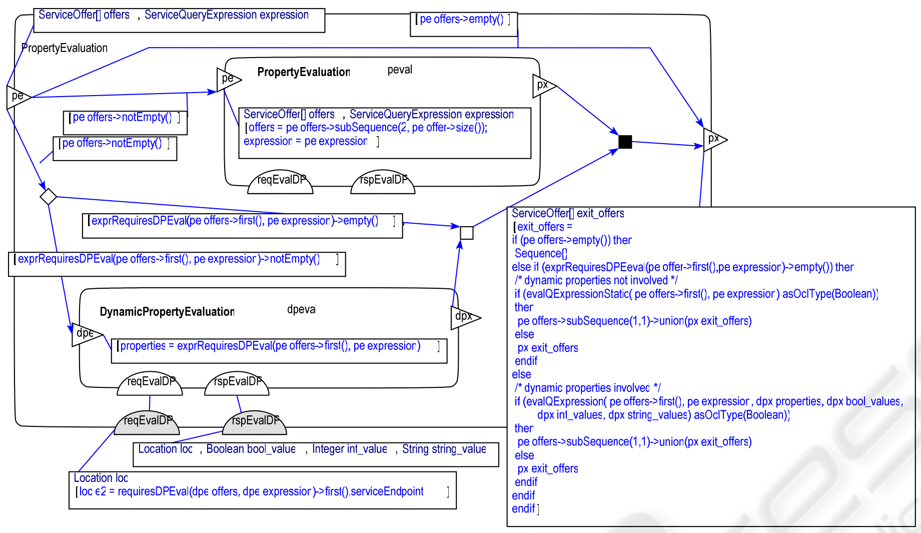
```
ServiceEndpoint[] serviceEndpoints
|serviceEndpoints =
   peval px exit_offers->collect(serviceEndpoint)
```

```
ServiceOffer[] offers , ServiceOfferId[] ids
|b_21 e offers =
   if (export) then e offers->including(export offer)
   else if (withdraw) then e offers->excluding(e offers->at(e ids->indexOf(withdraw id)))
   else if (reqServiceQuery) then e offers
   endif
b_21 e ids =
   if (export) then e ids->including(export id)
   else if (withdraw) e ids->excluding(export id)
   else if (reqServiceQuery) then e ids
   endif
```

Fig. 6. ServiceTrader behaviour.

The *rspServiceQuery* interaction returns the list of endpoints for the service offers in *exit_offers*. The list of current offers (*offers*) is updated in a recursive instantiation of the *ServiceTrader* behaviour: the occurrence of *export* results in the inclusion of the exported offer (*export.offer*) in *offers* and the occurrence of *withdraw* results in the exclusion of the offer. In Fig. 6, a diamond represents a choice and a square represents a disjunction of enabling relations.

Fig. 7 shows the *PropertyEvaluation* behaviour definition. This behaviour evaluates the service query expression for each service offer and is specified by recursive instantiation. A service offer is only included in *exit_offers* when the service query evaluates to *true* for that particular offer. When the evaluation of a service query requires the evaluation of dynamic service properties, the *DynamicPropertyEvaluation* behaviour is instantiated. Since the recursively instantiated *PropertyEvaluation* behaviour is directly enabled, this recursive instantiation pattern does not force a particular order for service property evaluation: all service properties are evaluated independently, and the results are combined with a conjunction (a filled black square in the ISDL notation).


ServiceOffer[] offers , ServiceQueryExpression expression
[pe offers->empty()]
PropertyEvaluation
pe
PropertyEvaluation peval
pe
px
ServiceOffer[] offers , ServiceQueryExpression expression
[offers = pe offers->subSequence(2, pe offer->size());
expression = pe expression]
[pe offers->notEmpty()]
[pe offers->notEmpty()]
reqEvalDP
rspEvalDP
px
[exprRequiresDPEval(pe offers->first(), pe expression)->empty()]
[exprRequiresDPEval(pe offers->first(), pe expression)->notEmpty()]
DynamicPropertyEvaluation dpeval
dpe
dpx
[properties = exprRequiresDPEval(pe offers->first(), pe expression)]
reqEvalDP
rspEvalDP
reqEvalDP
rspEvalDP
Location loc , Boolean bool_value , Integer int_value , String string_value
Location loc
[loc ϵ2 = requiresDPEval(dpe offers, dpe expression)->first().serviceEndpoint]

```
ServiceOffer[] exit_offers
[exit_offers =
if (pe offers->empty()) then
 Sequence{}
else if (exprRequiresDPEeval(pe offer->first(),pe expression)->empty()) then
 /* dynamic properties not involved */
 if (evalQExpressionStatic( pe offers->first(), pe expression ) asOclType(Boolean))
 then
  pe offers->subSequence(1,1)->union(px exit_offers)
 else
  px exit_offers
 endif
else
 /* dynamic properties involved */
 if (evalQExpression( pe offers->first(), pe expression , dpx properties, dpx bool_values,
       dpx int_values, dpx string_values) asOclType(Boolean))
 then
  pe offers->subSequence(1,1)->union(px exit_offers)
 else
  px exit_offers
 endif
endif
endif ]
```

Fig. 7. PropertyEvaluation behaviour.

Fig. 8 shows the *DynamicPropertyEvaluation* behaviour definition. This behaviour is also defined by recursive instantiation, using the same instantiation pattern that was used for *PropertyEvaluation*. For each dynamic property, two interactions occur: *reqEvalDP* and *rspEvalDP*. These interactions occur at the endpoint registered in the service offer as a dynamic property evaluator.

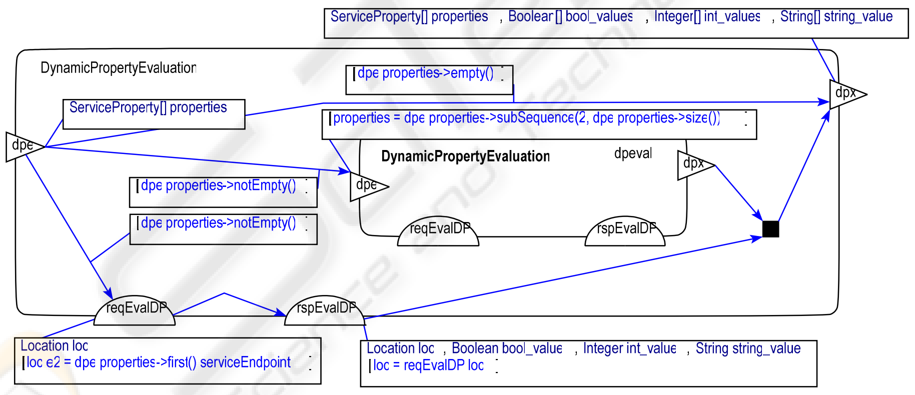

ServiceProperty[] properties , Boolean[] bool_values , Integer[] int_values , String[] string_value
DynamicPropertyEvaluation
[dpe properties->empty()]
dpx
ServiceProperty[] properties
[properties = dpe properties->subSequence(2, dpe properties->size())]
dpe
DynamicPropertyEvaluation dpeval
dpx
dpe
[dpe properties->notEmpty()]
[dpe properties->notEmpty()]
reqEvalDP
rspEvalDP
reqEvalDP
rspEvalDP
Location loc
[loc ϵ2 = dpe properties->first() serviceEndpoint]
Location loc , Boolean bool_value , Integer int_value , String string_value
[loc = reqEvalDP loc]

Fig. 8. DynamicPropertyEvaluation behaviour.

The following OCL definitions have been omitted here due to space limitations: *evalQExpression* and *evalQExpressionStatic*, which are used in *PropertyEvaluation* to determine whether an offer complies with a service expression; and *exprRequiresEval*, which is used select properties that must be evaluated in order to evaluate the expression. The complete trader specification can be found in [6]. All constraints in the specification are defined as follows: the left-hand side consists of the name of the (location or information) attribute being constrained; and the right hand side consists of a side-effect-free OCL expression. The expression determines the value of the constrained attribute. This simplifies significantly the evaluation of constraints in the simulation of the service behaviour.

4 Transformation Patterns

In this section, we discuss the transformation patterns related to the SOA trader abstract platform. As an example application we consider a printer service.

4.1 From Application Service Specification to Platform-Independent Application Design

We assume that an interaction *printReq* is defined at the application service specification level, which determines that some client has requested to print some document. In this example, the client of the printer service defines the maximum size of the queue it is willing to accept. This is done by using a combination of a value passing and value generation interaction (in accordance with the terminology of Section 3.2): the document is passed to the printer service and the size of the queue is determined possibly after consulting the queue length of many different printers, taking into consideration the interaction constraint of the maximum queue size imposed by the printer client. The actual size of the queue determines whether the interaction is successful or not. The use of this kind of interaction is only allowed at the application service specification level. Fig. 9 shows the *PrinterClient* and the *PrinterService* at the service specification level.

At the platform-independent application design level, the original interaction corresponds to a sequence of three (value passing) interactions: a request to the service trader, a response from the service trader and the actual interaction. Expressions on service properties in the query to the service trader are derived from information attributes and their constraints at the service specification level. This derivation requires marking of the service specification to indicate which information attributes should be used in the service query (in this case, the attribute *queueSize*). The interaction occurs at a service endpoint according to the response issued by the service trader. Fig. 9 also shows the *PrinterClient_* and the *PrinterService_* at the platform-independent application design level (the trader service is omitted because of space limitations). The *queueSizeReq* and *queueSizeRsp* are used to evaluate the queue size dynamic property.

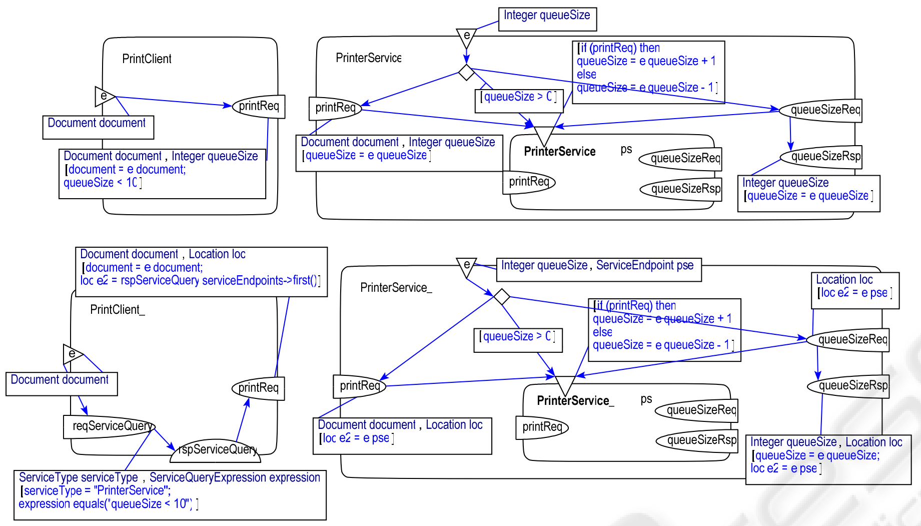

PrintClient
e
printReq
Document document
Document document , Integer queueSize
[document = e document;
queueSize < 10]
Integer queueSize
PrinterService
e
[if (printReq) then
queueSize = e queueSize + 1
else
queueSize = e queueSize - 1]
printReq
[queueSize > 0]
queueSizeReq
Document document , Integer queueSize
[queueSize = e queueSize]
PrinterService
ps
queueSizeReq
queueSizeRsp
printReq
queueSizeRsp
Integer queueSize
[queueSize = e queueSize]
Document document , Location loc
[document = e document;
loc e2 = rspServiceQuery serviceEndpoints->first()]
PrintClient_
e
Document document
printReq
reqServiceQuery
rspServiceQuery
ServiceType serviceType , ServiceQueryExpression expression
[serviceType = "PrinterService";
expression equals("queueSize < 10")]
Integer queueSize , ServiceEndpoint pse
PrinterService_
e
Location loc
[loc e2 = e pse]
[if (printReq) then
queueSize = e queueSize + 1
else
queueSize = e queueSize - 1]
[queueSize > 0]
queueSizeReq
printReq
PrinterService_
ps
queueSizeReq
queueSizeRsp
printReq
queueSizeRsp
Document document , Location loc
[loc e2 = e pse]
Integer queueSize , Location loc
[queueSize = e queueSize;
loc e2 = e pse]

Fig. 9. *DynamicPropertyEvaluation* behaviour.

The decision to implement the abstract *printReq* interaction as combination of a query and the actual print request may not be formally correct according to our refinement rules. This is because we cannot guarantee that the actual queue size at the time of the print request at the lower abstraction level is smaller than the maximum queue size, as prescribed in the most abstract specification. However, this implementation is an acceptable approximation if (i) the time between the *reqServiceQuery* and the *printReq* in behaviour *PrintClient_* is negligible compared with the rate at which jobs are submitted to the printer, and (ii) the SOA trader is capable of timely updating the dynamic properties.

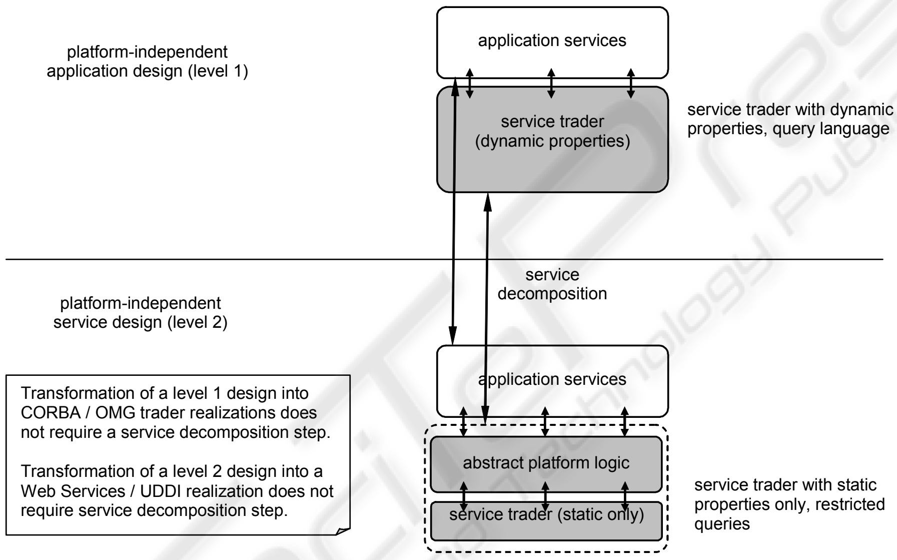
4.2 From Platform-Independent Service to Platform-Specific Service

In order to show the flexibility of the relation between the platform-independent application design level and the platform-specific application design we describe below a possible transformation of platform-independent application designs into two different middleware platforms, namely, Web Services and CORBA. These platforms differ significantly with respect to their support for service discovery.

CORBA provides a trader [17] that supplies a constraint language that allows one to define expressions that correspond to *ServiceQueryExpression* attribute values. In [6], a textual syntax for a *ServiceQueryExpression* has been defined such that any *ServiceQueryExpression* in this form is identical to an expression in the OMG trader constraint language. Furthermore, the OMG trader also supports dynamic service properties. A service exporter must implement the *DynamicPropEval* IDL interface [17]. This interface includes an *evalDP* operation, which receives as a parameter the property name and the required return type. The *evalDP* operation returns the value of the property.

In the case of Web Services technologies, service discovery is provided by UDDI [15]. UDDI does not support dynamic service properties and supports no query language, being able only to provide the values of static service properties (*tModels* [15]) to its clients.

A realisation of the trader service in CORBA is rather straightforward and does not require decomposition of the trader service. A realisation of the trader service in UDDI is more complex due to the differences in the support provided by UDDI and the trader service as specified in the abstract platform. We approach this by introducing a service decomposition step prior to realisation. Fig. 10 shows the two approaches to platform-specific realization. In the case of the CORBA realisation, only one platform-independent application design level is used (level 1 in Fig. 10). In the case of the Web Services/UDDI realization, both platform-independent application design levels 1 and 2 are used.


platform-independent application design (level 1)
application services
service trader (dynamic properties)
service trader with dynamic properties, query language
service decomposition
platform-independent service design (level 2)
application services
abstract platform logic
service trader (static only)
service trader with static properties only, restricted queries

Transformation of a level 1 design into CORBA / OMG trader realizations does not require a service decomposition step.

Transformation of a level 2 design into a Web Services / UDDI realization does not require service decomposition step.

Fig. 10. Realization of the SOA trader platform into two different platforms.

The abstract platform logic must bridge the gap between the trader service at the abstract platform and the service provided by a UDDI registry. Each service offer is registered as an entry in the UDDI registry. Given a query, the abstract platform logic uses the UDDI registry to retrieve all entries for a particular service type, evaluates the expressions (which may include dynamic property evaluation) and returns the list of service offers for which expressions evaluate to *true*. In order to support dynamic service properties, Web service endpoints that are used to evaluate dynamic properties must be registered as an additional *tModel*, which is present only for dynamic service properties.

5 Conclusions and Future Work

We have discussed how services can be modelled in a platform-independent manner, using a combination of a general-purpose behaviour modelling language (ISDL) with UML class diagrams and OCL constraints. The result is an abstract platform for platform-independent application designs based on the SOA principles. We have applied the modelling technique for the trader service, introducing the service discovery pattern at the platform-independent level. The trader service supports dynamic service properties and a simple constraint language for service queries.

We stress that the trader service specification in ISDL defines constraints on the interactions of a client with the trader, without prescribing any internal details of the trader. This is compatible with the service-oriented design principle that services only concern observable behaviour [22]. This gives us maximum flexibility for the further decomposition of the trader, as shown by the realisation of the service trader into a more rudimentary trader (UDDI, featuring no constraint language and no dynamic service properties). This realisation illustrates how target platform differences can be accommodated in the platform-specific realisation step. Further, our specification of the trader service is such that no particular strategy for evaluation of static or dynamic properties is implied. This allows different strategies to be adopted at platform-specific realisation level.

We have used ISDL to model the behavioural aspects of services for four main reasons. Firstly, ISDL supports a broad spectrum of abstraction levels which allows us to cover from service specification to service design seamlessly. Secondly, the concept of abstract interaction in ISDL enables us to capture service designs in a middleware-platform-independent manner (as shown in [2]). Thirdly, ISDL allows to capture causality relations between interactions without constraining the internal implementation of services. And, finally, conformance rules have been defined [23] which can be used to verify whether service designs respect service specifications.

Most approaches to MDA and SOA in literature ignore the description of the behaviour of individual services, specifying individual services solely based on messages exchanged (e.g., described in WSDL or UML class diagrams [10]), or focusing solely on the orchestration of multiple services (e.g., [12]). A consequence of this is that properties of the composition of services cannot be derived from specifications and specifications cannot be simulated. The modelling techniques we have discussed in this paper addressed both aspects.

We have focused on the behavioural aspects of the SOA trader abstract platform and we have not considered the typing system for the trader service. A natural extension of the work reported in this paper is the support for taxonomies and service typing rules.

We have used the Grizzle tool [9] to simulate the trader service specification. Further work on the tool support will involve integrating this tool with support for MOF QVT transformations [20], which will allow us to specify and execute the transformations discussed in this paper in generic model transformation tools. Currently some experiments with a transformation similar to that of section 4.1 have been reported in [6] using GReAT model transformations [1].

Acknowledgements

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

References

1. A. Agrawal, G. Karsai, A. Ledeczi, "An end-to-end domain-driven software development framework". In: Proc. 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'03). ACM Press (2003) 8–15
2. Almeida, J.P.A., Dijkman, R., Ferreira Pires, L., Quartel, D., van Sinderen, M.: Abstract Interactions and Interaction Refinement in Model-Driven Design. In: Proceedings Ninth IEEE EDOC Conference (EDOC 2005), IEEE Computer Society Press, Sept. (2005) 273–286
3. 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 Seventh IEEE Int'l Conf. on Enterprise Distributed Object Computing (EDOC 2003). IEEE Computer Society Press (2003) 112–123
4. Almeida, J.P.A., Dijkman, R., van Sinderen, M., Ferreira Pires, L.: Platform-Independent Modelling in MDA: Supporting Abstract Platforms, in Proceedings Model-Driven Architecture: Foundations and Applications 2004 (MDAFA 2004), Linköping University, Linköping, Sweden, (2004) 219–233. Revised version appeared in Lecture Notes in Computer Science, vol. 3599, Springer (2005) 174–188
5. Almeida, J.P.A. Dijkman, R. van Sinderen, M., Ferreira Pires, L.: On the Notion of Abstract Platform in MDA Development, In: Proc. 8th IEEE Int'l Conf. on Enterprise Distributed Object Computing (EDOC 2004), IEEE Computer Society Press, Sept. (2004) 253–263
6. Almeida, J.P.A., Iacob, M.E., Jonkers, H., Quartel, D.: Platform-Independent Modelling of Service Infrastructure Components, Freeband A-MUSE/D1.6, TI/RS/2005/078, Telematica Instituut, Enschede, The Netherlands (2005); https://doc.telin.nl/dscgi/ds.py/Get/File-59319
7. Dijkman, R.M.: Consistency in Multi-Viewpoint Architectural Design, Ph.D. thesis, University of Twente, The Netherlands (2006)
8. Erl, T.: Service-oriented architecture: Concepts, technology, and design. Prentice-Hall (2005)
9. Grizzle, http://isdl.ctit.utwente.nl/tools/grizzle
10. Grønmo, R., Skogan, D., Solheim, I., Oldevik, J.: Model-driven Web Services Development. In Proceedings IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE-04), Taipei, Taiwan (2004) 42–45
11. Kutvonen, L.: Achieving Interoperability through ODP Trading Function, In: Proc. 2nd Int'l Symposium on Autonomous Decentralized systems (ISADS 1995), IEEE Computer Society Press, Apr. (1995) 63–69
12. Mantell, K.: From UML to BPEL, Model Driven Architecture in a Web services world, IBM (2005) http://www-128.ibm.com/developerworks/webservices/library/ws-uml2bpel/
13. ISDL home, http://isdl.ctit.utwente.nl/
14. ITU-T / ISO: ODP Trading Function: Specification, ITU-T Recommendation X.950 | IS 13235-1 (1997)
15. OASIS: OASIS - Committees - OASIS UDDI Specifications TC; http://oasis-open.org/committees/uddi-spec/doc/tcspecs.htm

16. Object Management Group: MDA-Guide, Version 1.0.1, omg/03-06-01 (2003)
17. Object Management Group: Trading Object Service Specification, V1.0, formal/00-06-27 (2000)
18. Object Management Group: Unified Modelling Language: Object Constraint Language version 2.0*,* ptc/03-10-04 (2003)
19. Object Management Group: UML 2.0 Superstructure, ptc/03-08-02 (2003)
20. Object Management Group: 2nd revised submission to the MOF 2.0 Q/V/T RFP, ad/05-03-02 (2005)
21. Papazoglou, M.P., Georgakopoulos, D.: Service-oriented computing. In: Communications of the ACM, Vol. 46, No. 10 (2003) 24–28
22. Quartel, D., Dijkman, R., van Sinderen, M.: Methodological support for service-oriented design with ISDL. In: Proceedings of the 2nd International Conference on Service-Oriented Computing (ICSOC) (2004) 1–10
23. Quartel, D.: Action relations Basic design concepts for behaviour modelling and refinement, Ph.D. thesis, University of Twente, Enschede, The Netherlands (1998)
24. Quartel, D., Ferreira Pires, L., van Sinderen, M., Franken, H., Vissers, C.: On the role of basic design concepts in behaviour structuring. In: Computer Networks and ISDN Systems, Vol. 29, No. 4 (1997) 413–436
25. Quartel, D. Ferreira Pires, L., van Sinderen, M.: On Architectural Support for Behaviour Refinement. In: Distributed Systems Design, Journal of Integrated Design and Process Science, Vol. 6, No. 1. Society for Design and Process Science (2002)
26. Vinoski, S.: Service Discovery 101, in IEEE Internet Computing, IEEE Computer Society, Vol. 7, No. 1 (2003) 69–71
27. World Wide Web Consortium: SOAP Version 1.2 Part 1: Messaging Framework, W3C Proposed Recommendation (2003); http://www.w3.org/TR/soap12-part1
28. World Wide Web Consortium: Web Services Description Language (WSDL) 1.1, W3C Note (2001); http://www.w3.org/TR/wsdl