Formalizing High-Level Service-Oriented Architectural Models Using a Dynamic ADL

Marcos López-Sanz, Carlos E. Cuesta, and Esperanza Marcos

Kybele Research Group
Rey Juan Carlos University
Mostoles – 28933 Madrid, Spain
{marcos.lopez,carlos.cuesta,esperanza.marcos}@urjc.es

Abstract. Despite the well-known advantages of applying the MDA approach, particularly when applied to the development of SOA-based systems, there are still some gaps in the process that need to be filled. Specifically, when modelling the system at the PIM level, we have an architectural description at a high level of abstraction, as it must only comprise technologically independent models. But this architecture cannot be directly validated, as we must transform it into a PSM version before being able to execute it. In order to solve this issue, we propose to formalize the architectural model using Domain Specific Language, an ADL which supports the description of dynamic, adaptive and evolvable architectures, such as SOA itself. Our choice, π-ADL, allows for the definition of executable versions of the architecture; and therefore providing this specification implies having a prototype of the system at the PIM level. This appears as a perfect way of getting an executable yet completely technology neutral version of the architecture. We illustrate this by discussing a real-world case study, in which a service-oriented messaging system is modelled at the PIM level and then specified using its π-ADL counterpart; the result can then be used to validate the architecture at the right abstraction level.

Keywords: Service-Oriented Architecture, Model-Driven Architecture, PIM-level modelling, π-ADL.

1 Introduction

Service orientation has established as leading technological trend due to its advantages for cross-organization integration, adaptability and scalability. As the Service-Oriented Computing (SOC) paradigm [2] is largely considered as the *de facto* solution for emerging information society challenges, many application areas are taking advantage of services. They range from the field of Software Architecture to the definition of software development processes; and, in particular, as a foundation for environments in which dynamism and adaptation are key concerns.

Taking a deeper look at the Software Engineering field we found strategies that benefit from and contribute to the SOC paradigm. The Model-Driven Architecture (MDA) proposal [9], in particular, has been used to develop methods [4] using the principles of SOC and also for the implementation of SOA solutions [3].

However, and despite the well-known advantages of MDA (separation in abstraction levels, definition of automatic transformations between models, model adaptability to multiple domains, etc.), the model-driven approach lacks the ability to define *early* executable versions of the system. Taking into account the separation in CIM, PIM and PSM levels stated by the MDA proposal, it is only at the PSM level when the features of a specific technology are considered. Therefore, it is not possible to get a working prototype until reaching that abstraction level. In this paper we study this problem in the context of SOA-based architectural modelling and within a model-driven methodological framework called MIDAS [2].

The architecture of a software system comprises "*all the components of a system, their relationships to each other and the environment and the principles governing its design and evolution*" [6]. In that sense, the architecture is considered to be a concrete view of the system which corresponds, essentially, with an abstraction of its structure and main behaviour [7].

In previous work we defined the foundations of architectural models supporting all the features of Service-Oriented Architectures (SOA) at the PIM level of abstraction. Considering the role of the architecture in a model-driven framework and the necessity of an early executable version of the system, we have found that using a language from the architectural domain of research might be the best option for overcome those needs. In particular, we have focused π-ADL [10] as a Domain Specific Language (DSL) for the representation of our architectural models.

By providing a correspondence between the concepts gathered in the architectural models at PIM level and their counterparts in a language such as π-ADL, we ensure that, on the one hand, the architectural models are *sound* and, on the other hand, that the architect has at his disposal a toolset for creating a valid executable prototype of the system in early stages of the development process.

The structure of the paper is as follows: Section 2 gives an overview of the basic concepts considered in this paper: π-ADL, MDA and SOA. Section 3 presents a case study used to illustrate the benefits of using π-ADL for describing SOA architectures that represent architectural models at the PIM level of abstraction. Finally, Section 4 discusses the main contributions of this article and some of the future works.

2 Previous Concepts

This article is part of a much broader research effort: the refinement of MIDAS [2], a software development framework based on the MDA principles. The model architecture of this methodology is divided into 3 orthogonal dimensions, describing the abstraction levels (vertical axis), core concerns (horizontal) and crosscutting aspects (transversal); the latter includes the architectural model itself.

The architectural description must cover both PIM (Platform-Independent Model) and PSM (Platform-Specific Model) levels of abstraction, as defined in MDA. As it has been explained in previous work [7], the main reason to make this is that MIDAS follows an ACMDA (*Architecture-Centric Model-Driven Architecture*) approach: it defines a method for the development of information systems based on models and guided by the architecture.

With an architectural view of the system at PIM level, we facilitate the establishment of different PSM-level models according to the specific target platform which are derived from a unique PIM model. However, this has as main drawback the impossibility of having a precise executable version of the system.

2.1 Specification of Service-Oriented Architectural Models

This subsection covers the description of the concepts supporting the architectural modelling and the relationships among them. These concepts are gathered in the metamodel used to define service-oriented architectural models at PIM level in MIDAS. The service metamodel containing the foundations of that model can be seen in Figure 1. Next each of the concepts is briefly explained.

Service Providers

Generally speaking, a SOA is built upon independent entities which provide and manage services. Service providers can be classified into two main groups:

- ✓ **Inner service providers**, which are internal organizations to the system designed. They can be also understood as the part of the software solution whose inner elements are being designed in detail.
- ✓ **Outer service providers,** which are external entities containing services which collaborate with the system to perform a specific task of value for the system but which are not under its control or whose internal structure is not known or valuable for the development of the current architectural model.

The relationship between two service providers appears in the moment that a business need arises and the services involved in its resolution belong to each of them. Because the interconnected elements represent business entities, the relation among them is understood as a '**business contract**'.

Service Description: Identity, Operations and Roles

Our vision of the service concept, at the PIM level, is aligned with that of the OASIS reference model for services [8]. That is, therefore, *the atomic artefact within the architectural modelling that allows the practical support for the system features and business processes identified in higher abstraction layers during the development process*. The main elements that allow the description of a service at the PIM level of abstraction are: the **SERVID,** a **set of operations**, and a **service role**. With these three elements it is enough to clearly model a service within a concrete architectural configuration.

Service Interaction: Contracts and Interaction Patterns

Service interaction is based on the tacit agreement to act in a concrete manner gathered under a contract signed by the participant services.

- ✓ **Service contracts.** Services relate, communicate and interact with each other according to agreed contracts. In the architectural description of the service models, these 'contracts' are understood as connectors, specifying point-to-point relationships between the services that 'sign' those contracts.

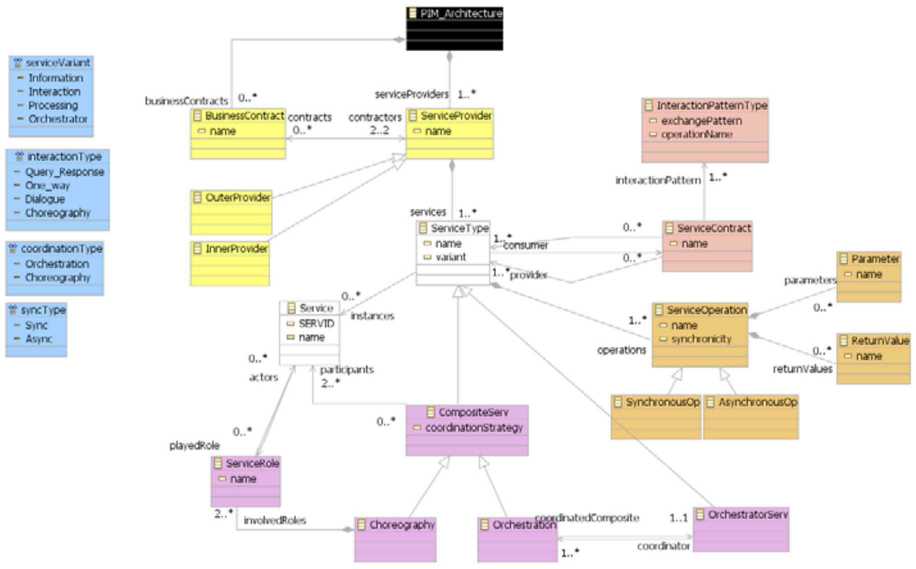

Fig. 1. PIM-level service architectural metamodel

- ✓ **Interaction Patterns.** These interactions are defined as pairs '*operation name-interaction kind*' establishing a connection between the set of operations available from the provider service and the kind of exchange pattern that will be followed when using that operation. These patterns are reduced to four alternatives. '*One-way*', '*Query/Response*', '*Dialogue*' (in which the concrete protocol can be complex) and '*Choreography*' which will be used in scenarios where several services agree to collaborate to reach a common goal.

Service Composition: Orchestrations and Choreographies

Composition within SOA is accomplished by means of an interactive collaboration among its participants according to a coordination scheme and without constraining the independence of the composed elements. The classification of service composition alternatives is done used to build up the composition:

- ✓ **Orchestration.** This kind of coordination is founded upon the idea of having a special service in charge of directing the whole compound element. This special service (*OrchestratorServ* in the metamodel) knows the flow of service consumptions that must be done to accomplish the functionality desired.
- ✓ **Choreography.** This kind of composition represents an interacting environment among equivalent services. This "equivalence" means that there is no service mastering over any other or directing the flow of information.

2.2 π-ADL at a Glance

π-ADL [10] is a language designed for defining software architectures and is formally founded on the higher-order typed π-calculus. π-ADL.NET is a compiler tool for π-ADL, and was predominantly used to validate the case study presented in section 3.

In a π-ADL program, the top level constructs are *behaviours* and *abstractions*. Each behaviour definition results in a separate execution entry point, meaning that the program will have as many top level concurrent threads of execution as the number of behaviours it defines. Abstractions are reusable behaviour templates and their functionality can be invoked from behaviours as well as other abstractions.

The body of a behaviour (or an abstraction) can contain *variable* and *connection* declarations. Connections provide functionality analogous to channels in π-calculus: code in different parts of behaviours or abstractions can communicate synchronously via connections. Connections are typed, and can send and receive any of the existing variable types, as well as connections themselves. Sending a value via a connection is called an output-prefix, and receiving via a connection is called an input prefix.

The *compose* keyword serves the purpose of creating two or more parallel threads of execution within a program and corresponds to the concurrency construct in π-calculus. Another important π-ADL construct is the *choose* block. Only one of the sub-blocks inside a choose block is executed when execution passes into it. Finally, to provide the equivalent of the π-calculus replicate construct, π-ADL provides the *replicate* keyword.

3 Case Study

The selected case study emulates the functionality of a SMPP (Short Message Peer-to-Peer Protocol) [12] gateway system by means of services. SMPP is a telecommunications industry protocol for exchanging SMS messages between SMS peer entities. We focus on the following building blocks and functionalities:

- *Reception Subsystem*: its main purpose is to receive sending SMS requests directly from the user. It contains a single service (*ReceptionService*) offering several useful operations for the user to send SMS texts.
- *Storage Subsystem*: this subsystem stores information related to clients and SMS messages. It comprises two services:
  - *SecureDataService*: performs operations requiring a secure connection.
  - *SMSManagerService*: in charge of managing all the information related to SMS messages (such as status, SMS text) and creating listings and reports.
- *SMS Processing Subsystem*: this subsystem is in charge of retrieving, processing and sending the SMS texts and related information to the specialized SMS server.
  - *SMSSenderService*: Retrieves SMS texts from the Storage Subsystem and sends them to Short Message Service Centres (SMSC).

    DirectoryService: The main task performed by this service is to return the service identifier of the *SMSCenterService* which has to be used to send a SMS to a specific recipient.
- *SMSC (Short Message Service Centre):* this entity represents specialized SMS servers capable to send the SMS texts to a number of recipients. Its functionality is enacted by one service (*SMSCenterService*).

Next, we present a partial formalization of the architectural model with π-ADL, emphasizing the aspects of that provide an adequate solution for our system as well as explaining how the structures and principles of π-ADL are adapted to our vision of PIM-level service architecture:

Representation of a service and its operations. Services represent computing entities performing a specific behaviour within the system architecture and thus they are specified by means of π-ADL abstractions (see Listing 3.1 for the architectural specification of the *ReceptionService*).

Every service abstraction defines its own communication channels through input and output connections. The data acquired and sent by these connections comprises a description of the operation and the data associated to that message. Depending on the operation requested, the service abstraction will transfer the control of the execution to the corresponding operation. The only behaviour associated with the service abstraction is, therefore, that of redirecting the functionality request to the corresponding operation and sending back returning values if any.

Operations, in turn, are also specified by means of abstractions as they encapsulate part of the functionality offered by services. Like any other abstraction used in the description of the service architecture, operation abstractions will receive the information tokens through connections, sending back an answer when applicable.

```
value ReceptionService is abstraction () {
// omitted variable definition
if (input::operation == "sendSMS") do {
  via SendSMS send input::data where {resultConn renames resultConn};
  via resultConn receive result;
  compose { via outConn send result; and  done;    }   }}
```

Listing 3.1. Specification of the *ReceptionService*

In π-ADL communication through the connections is performed synchronously. This means that communication with operations is synchronous. Therefore, the semantics associated with the asynchronous operations are lost since the abstraction will be blocked in a send operation until any other abstraction in the architecture perform a receive operation over that channel. In order to model asynchronous operations, the specification can be placed in one of the sub-blocks of a compose block, with the second sub-block returning immediately with the `done` keyword.

Representation of contracts. As stated previously, services relate and communicate through contracts. Within the architecture these contracts are active connectors in charge of enabling the message exchange between services according to a specific pattern, represented by means of the programmatic specification of a state machine. Similarly, connectors in π-ADL are represented by means of abstractions.

In a static service environment, in which contracts between services are established at design time, all the information needed by a contract to correctly fulfil its behaviour (message exchange pattern and contractors) is defined and initialized internally within the contract abstraction when the system starts. In dynamic environments however, this is normally accomplished by transferring all the information through the channel opened simultaneously when the abstraction is executed. In both cases, the contract is able to perform the behaviour needed to transfer data requests and results from one service to another from that information.

Listing 3.2 depicts part of the analysis of a state of the message pattern execution. In it, it is shown how, in order to send anything to one of the services connected through the `Shipping` contract, a compose structure should be used: first to send the data through the connection and second to execute the abstraction and unify the connections.

When executing the specification of a service architecture with π-ADL any behaviour defined is carried out as an independent thread of execution. However, in order to be able to perform a coordinated and joint execution, the different abstractions must be linked. Because of the dynamic nature of the service architectures, contract abstractions can be reused as the instances of the services they communicate can vary during the lifecycle of the system. In order to achieve this behaviour, contracts (or more appropriately abstractions performing the contract role) must be able to dynamically instantiate the channel that they have to use to send or receive the data transferred in each moment. To deal with this issue π-ADL defines the `dynamic(<connection_name>)` operator. This operator represents one of the main advantages for dynamic architecture specification since π-ADL allows the transference of connections through connections.

```
...
if (state::via_SERVID == "S") do{
 compose {
   via outConnectionS send inData;
 and
   via dynamic(input::ServConnGroup(0)::SERVID) send Void
   where {outConnectionS renames inConn,inConnectionS renames outConn};
 }}else do{   via outConnectionC send inData; }
...
```

Listing 3.2. Fragment extracted from the *Shipping* contract

Representation of dynamism. In our case study, In order to send the SMS messages stored on the database, it is necessary to know which specific service should be used. To achieve that behaviour it is essential to be aware of the existence of a specialized service attending to requests from services asking for other services to perform tasks with specific requirements. This represents a dynamic environment since it is necessary to create a communication channel that did not exist at design time but is discovered when the system is in execution (i.e. when the *SMSManagerService* must send the SMS texts to a concrete *SMSCenterService*).

In a service-oriented environment the dynamicity may occur in several scenarios: when it is necessary to create a new contract between services or when the new element to add is another service. In those cases it is mandatory to have a special service in charge of performing the usual operations that occur in dynamic environments, i.e. *link*, *unlink*, *create* and *destroy* of contract abstractions. This service will be the *DirectoryService* that appears in Figure 2.

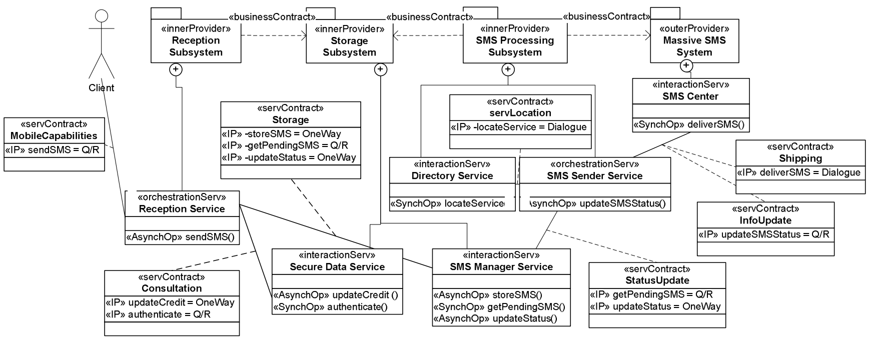
«businessContract»
«businessContract»
«businessContract»
«innerProvider» Reception Subsystem
«innerProvider» Storage Subsystem
«innerProvider» SMS Processing Subsystem
«outerProvider» Massive SMS System
Client
«interactionServ» SMS Center
«SynchOp» deliverSMS()
«servContract» MobileCapabilities
«IP» sendSMS = Q/R
«servContract» Storage
«IP» -storeSMS = OneWay
«IP» -getPendingSMS = Q/R
«IP» -updateStatus = OneWay
«servContract» servLocation
«IP» -locateService = Dialogue
«interactionServ» Directory Service
«SynchOp» locateService
«orchestrationServ» SMS Sender Service
«SynchOp» updateSMSStatus()
«servContract» Shipping
«IP» deliverSMS = Dialogue
«servContract» InfoUpdate
«IP» updateSMSStatus = Q/R
«orchestrationServ» Reception Service
«AsynchOp» sendSMS()
«interactionServ» Secure Data Service
«AsynchOp» updateCredit ()
«SynchOp» authenticate()
«interactionServ» SMS Manager Service
«AsynchOp» storeSMS()
«SynchOp» getPendingSMS()
«AsynchOp» updateStatus()
«servContract» Consultation
«IP» updateCredit = OneWay
«IP» authenticate = Q/R
«servContract» StatusUpdate
«IP» getPendingSMS = Q/R
«IP» updateStatus = OneWay

Fig. 2. Architectural model of the SMPP case study

```
  value locateService is abstraction (
  locationConn:view[inConn:Connection[view[operation:String,data:any]],
  input : view[operation : String, data : any],
  connSMSCenter: Connection[view[operation : String, data : any]]]){
        requestData = locationConn::input::data;
  compose       {
  // full description of the message exchange pattern omitted
  // prepare and create connector: Shipping
                via Shipping send contractStartInfo;
        and
                clientDataResponse::contractConn = shContractClientConn;
  // info about the service operations and semantics of the service
                clientDataResponse::servMetaInfo = "";
                clientResponse::data = clientDataResponse;
                clientResponse::operation = "locateService";
                via locationConn::inConn send clientResponse;
        and     done;         }}
```

Listing 3.3. Specification of the *locateService* operation

Representation of service composition. Coordination among services can be achieved by defining choreographies or orchestrations. Choreographies can be formalized with π-ADL by means of shared connections. Orchestrations, in turn, depend mostly on the code specified inside a unique abstraction belonging to a service playing the role of coordinator of the composition.

In our case study a service taking the orchestrator role is the SMS Sender Service which coordinates the access to the storage subsystem (using the SMS Manager service), the retrieval of the information of the concrete *SMSCenterService* to be used to send the SMS texts by invoking the Directory Service and finally the *SMSCenterService* to complete the desired functionality.

4 Conclusions and Future Works

MDA is one of the current leading trends in the definition of software development methodologies. Its basis lies in the definition of model sets divided in several abstraction levels together with model transformation rules. Observing the principles of the SOC paradigm and its inherent features for system integration and interoperability it is possible to accept MDA as a suitable approach for the development of SOA solutions.

While defining model-driven development frameworks, and MDA-based in particular, the architecture has been demonstrated to be the ideal source of guidance of the development process since it reflects the structure of the systems embedded in its components, the relations among them and their evolution during the lifecycle of the software being developed. In the case of MIDAS, we have defined UML metamodels for the PIM-level view of the architecture.

In this work, and in order to solve the initial lack of early prototypes in MDA-based developments, we have proposed to give a formal definition of the system architecture by means of an ADL. Specifically we have chosen π-ADL because of its support for representing dynamic and evolvable architectures as well as the largely faithful compiler tool available for this language. Moreover, by using a formal representation of the system we can use mathematical formalisms to validate the UML models created for each of the abstraction levels defined within MIDAS.

There are many research lines that arise from the work presented in this paper. One research direction is, given the already defined UML notation and metamodel for the π-ADL language, the definition of transformation rules between the UML metamodel of the architecture at PIM-level and that of the π-ADL language. More research lines include the refinement of the language support for specific SOA aspects such as the definition of choreographies, dynamic and evolvable environments requiring the representation of new types of components and connectors within the system architecture, etc.

Acknowledgements

This research is partially granted by projects MODEL-CAOS (TIN2008-03582) and AT (Agreement Technologies) (CONSOLIDER CSD2007-0022), from the Spanish Ministry of Science and Technology, and by COST AT (COST Action IC0801) from European Union RTD Framework Programme.

References

[1] Broy, M.: Model Driven, Architecture-Centric Modeling in Software Development. In: Proceedings of 9th Intl. Conf. in Engineering Complex Computer Systems (ICECCS 2004), pp. 3–12. IEEE Computer Society, Los Alamitos (April 2004)

[2] Cáceres, P., Marcos, E., Vela, B.: A MDA-Based Approach for Web Information System Development. In: Workshop in Software Model Engineering (2003), `http://www.metamodel.com/wisme-2003/` (retrieved March 2007)

[3] De Castro, V., López-Sanz, M., Marcos, E.: Business Process Development based on Web Services: A Web Information System for Medical Images Management and Processing. In: Leymann, F., Zhang, L.J. (eds.) Proceedings of IEEE International Conference on Web Services, pp. 807–814. IEEE Computer Society, Los Alamitos (2006)

[4] De Castro, V., Marcos, E., López-Sanz, M.: A Model Driven Method for Service Composition Modeling: A Case Study. International Journal of Web Engineering and Technology 2(4), 335–353 (2006)

[5] Gomaa, H.: Architecture-Centric Evolution in Software Product Lines. In: ECOOP'2005 Workshop on Architecture-Centric Evolution (ACE 2005), Glasgow (July 2005)

[6] IEEE AWG. IEEE RP-1471-2000: Recommended Practice for Architectural Description for Software-Intensive Systems. IEEE Computer Society Press (2000)

[7] López-Sanz, M., Acuña, C., Cuesta, C.E., Marcos, E.: Defining Service-Oriented Software Architecture Models for a MDA-based Development Process at the PIM level. In: Proceedings of WICSA 2008, Vancouver, Canada, pp. 309–312 (2008)

[8] Marcos, E., Acuña, C.J., Cuesta, C.E.: Integrating Software Architecture into a MDA Framework. In: Gruhn, V., Oquendo, F. (eds.) EWSA 2006. LNCS, vol. 4344, pp. 127–143. Springer, Heidelberg (2006)

[9] OASIS: Reference Model for Service Oriented Architecture (2006), Committee draft 1.0. from http://www.oasis-open.org/committees/download.php/16587/wd-soa-rm-cd1ED.pdf (retrieved Febuary 2007)

[10] OMG: Model Driven Architecture. In: Miller, J., Mukerji, J. (eds.) Document No. ormsc/2001-07-01 , http://www.omg.com/mda (retrieved May 2006)

[11] Oquendo, F.: π-ADL: An Architecture Description Language based on the Higher Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures. ACM Software Engineering Notes (3) (May 2004)

[12] Papazoglou, M.P.: Service-Oriented Computing: Concepts,Characteristics and Directions. In: Proc. of the Fourth International Conference on Web Information Systems Engineering (WISE 2003), Roma, Italy, December 10-12, pp. 3–12 (2003)

[13] SMPP Forum: SMPP v5.0 Specification, http://www.smsforum.net/ (retrieved September 2007)