A Model-Driven Approach to Service Orchestration

Philip Mayer, Andreas Schroeder, Nora Koch

Institute for Informatics, Ludwig-Maximilians-Universität München
Oettingenstr. 67, 80538 München, Germany

{mayer, schroeda, kochn}@pst.ifi.lmu.de

Abstract

Software systems based on Service-Oriented Architectures (SOAs) promise high flexibility, improved maintainability, and simple re-use of functionality. A variety of languages and standards have emerged for working with SOA artifacts; however, service computing still lacks an effective and intuitive model-driven approach starting from models written in an established modeling language like UML and, in the end, generating comprehensive executable code. In this paper, we present a conservative extension to the UML2 for modeling service orchestrations at a high level of abstraction, and a fully automatic approach for transforming these orchestrations down to the well-known Web Service standard BPEL.

1. Introduction

With the introduction of the Service-Oriented Architecture (SOA), formerly proprietary software systems are being opened up and made available as services. On top of these services, business processes and technical workflows are being (re-)implemented as compositions of services, which has come to be known as service orchestration. Service computing has quickly been embraced by both academy and industry, as it promises highly flexible software systems, simple re-use of functionality, and improved maintainability.

While model-driven approaches are already in use for other programming paradigms, service-oriented design still falls short of effective and comprehensive domain-specific modeling and code generation tools. In order to support software engineers with intuitive and easy to adopt design and implementation techniques for service-oriented software, we propose (1) to extend the reach of UML to the modeling of SOA systems, and (2) to exploit so-designed models for creating running systems, in particular through code generation.

UML is a well-known and mature language for modeling software systems, however it is strenuous right now to model SOA artifacts with UML2, as native support for services and service orchestration concepts is missing. We therefore introduce a UML2 extension for SOA – called the UML4SOA profile – which is a high-level domain specific language for modeling service orchestrations as an extension of UML2 activity diagrams (Section 2). Based on this profile, we have defined transformation mechanisms from UML4SOA models to executable languages like BPEL, Java, and Jolie [5]. Here, we introduce our transformation to BPEL (Section 3). The paper concludes with a review of related work (Section 4) and a summary (Section 5).

2. Modeling Service Orchestration in UML

Modeling service orchestrations in plain UML reveals several important shortcomings, leading to the introduction of (unreadable) technical constructs. In particular, the following key distinguishing concepts of service compositions are missing: modeling of partners of a service; message passing among requester and provider of services, long-running transactions, compensation, and events. For example:

- It is not possible to restrict the set of valid callers of certain services – as needed e.g. to ensure that only specific external services can invoke an action – on an UML *AcceptCallAction*. All restrictions must be implemented manually.
- Event handlers are not directly supported. For example, temporally enabled event handlers must be manually disabled using technical constructs. Russel et al. [7] suggest using *InterruptibleActivity-Regions* containing the tasks to disable, and interrupting edges for normal task completion. However, using these technical constructs makes the diagrams harder to understand.
- Similarly, compensation handling is not directly included. Compensation for an activity cannot be associated directly with it, but must be programmed within explicit compensation logic. Modeling the compensation logic for more than one compensable activity is a tedious and error prone task [10].

Due to these shortcomings, modeling service orchestrations with plain UML is a cumbersome task. At the same time, the resulting UML models are difficult to transform to other languages, as the patterns used to handle the issues named above need to be recognized appropriately.

2.1. UML4SOA

To overcome these difficulties, we extend the UML2 with service-specific model elements, providing special elements for service interactions, long running transactions and their compensation as well as event- and exception handling.

Our UML2 extension is built on top of the Meta Object Facility (MOF) metamodel [6] and defined as a conservative extension of the UML2 metamodel. For the new elements of this metamodel, a UML profile is created using the extension mechanisms provided by UML. The principle followed is that of minimal extension, i.e. to use UML constructs wherever possible and only define new model elements for specific service-oriented features and patterns making diagrams simple, concise, and easy to understand.

The metamodel depicted in Figure 1 shows the model elements we introduce to UML2 activity diagrams. A brief description of the most distinguishing stereotypes is given below.

- An *orchestration* is a specialized UML *Activity* for modeling service orchestrations. Each orchestration contains a root scope.
- A *scope* is a UML *StructuredActivityNode* that contains arbitrary *ActivityNodes*, and may have an associated compensation handler.
- Specialized actions have been defined for sending and receiving data. In particular, a *send* is an UML *CallBehaviourAction* that sends a message; it does not block. A *receive* is a UML *AcceptCallAction*, receiving a message, which blocks until a message is received.
- Service interactions may have interaction pins for sending or receiving data. In particular, *lnk* is an UML *Pin* that holds a reference to the service involved in the interaction, *snd* is a *Pin* that holds a container with data to be sent, and *rcv* is a *Pin* that holds a container for data to be received.
- Finally, specialized edges connect scopes with handlers. For example, *compensation* is a UML *ActivityEdge* to add compensation handlers to actions and scopes.

Our profile also contains elements for event- and exception handling; they are not included here for lack of space. For a complete overview, see [3].

Figure 2 shows an example orchestration scenario – a typical SOA example of a ticket booking service which works in-between a theater and a customer. It is important to note that the metamodel introduced above only defines the new elements required for service orchestration and leaves everything else to the UML: the diagram shows how elements from the UML (in this case, actions, structured activity nodes, and branches) have been combined with new elements for service orchestrations (in this case, stereotypes for scopes and service interactions as well as new elements for compensation).

In general, using the UML4SOA constructs greatly reduces the number of technical constructs needed to model key SOA concepts like service interactions, compensation, and event handling. In the example, service interactions are specified by stereotypes; complex data flow edges have been replaced by pins with incoming and outgoing stereotypes; and

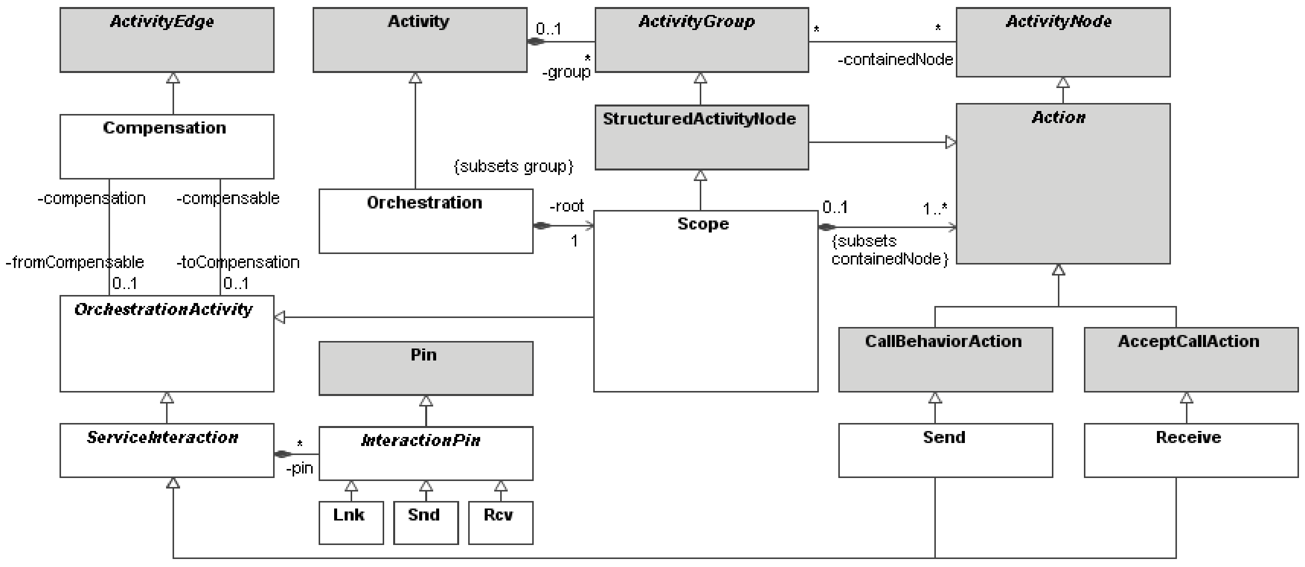

Figure 1: UML4SOA Metamodel (UML metaclasses in grey)

compensation handling has been introduced by using a specialized edge.

The value of the produced diagrams is increased for both human reading and automatic processing: the former profits from the concise and explicit – but minimalistic – labeling of constructs, while the latter profits from the simpler model structure.

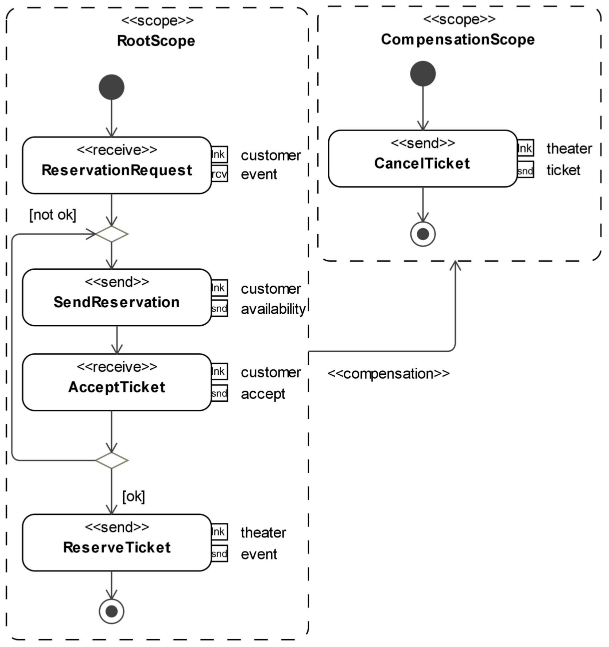

Figure 2: Orchestration Example

The UML2 profile defined above is available as a plug-in for IBM's Rational Software Architect as well as the MagicDraw UML modeler. Both are available for download from our website, *www.pst.ifi.lmu.de/ projekte/uml4soa/*.

3. Code Generation

The UML models we have introduced above already have great value for communicating the orchestration workflow; however, they are not yet executable. Therefore, we have implemented code generators for several target languages, among them BPEL/WSDL, the Jolie language, and Java code. In this section, we will detail our transformation to BPEL. The transformers are available as plug-ins for Eclipse and are likewise available from the website mentioned previously.

3.1. Transforming to BPEL

Although BPEL allows several alternatives for modeling processes, using structured activities like *repeatUntil*, *while*, or *if/else* is commonly favored over a graph-based approach due to better readability. Indeed, with BPEL 2.0 there seems to be a shift towards a more structured approach to the modeling of processes, as more structuring activities have been added.

Transforming to a structured BPEL process, however, poses some problems as the source models (activity diagrams) use graph structures. For example, branches and loops are modeled using the same elements (decisions/merges); their meaning therefore needs to be inferred from the context, i.e. the number of edges connected to them and their position within the control flow. Thus, the UML4SOA model transformer employs a depth-first rule-based approach to the conversion, which uses a partitioning algorithm to group UML activity diagram nodes for implementation by a certain BPEL structured activity.

```
<scope name="RootScope">
 <compensationHandler>
    ...
 </compensationHandler>

 <sequence>
   <receive name="ReservationRequest"
     operation="ReservationRequest"
     partnerLink="customer"
     variable="event"/>

   <repeatUntil>
      ...
   </repeatUntil>

   <invoke name="ReserveTicket"
     inputVariable="event"
     operation="ReserveTicket"
     partnerLink="theater"/>
 </sequence>
</scope>
```

Figure 3: Generated BPEL Code

There are three types of partitions which need to be identified in the UML source:

- Branches. Branching the control flow is modeled in UML with *decision* and *merge nodes*. In BPEL, branching is modeled with an *if* structured activity which may contain *elseif* branches for alternatives.
- Loops. We assume loops in the control flow to be modeled in UML with *merge* and *decision nodes*, with one control path leading from the decision at the end to the merge at the beginning. The equivalent BPEL construct for this is the *RepeatUntil* loop, which runs at least once.
- Parallel flows. Parallel execution is modeled in UML by using *fork* and *join nodes*. In BPEL, parallel flow is handled through the *flow* construct.

Besides these induced partitions, we also exploit explicit structuring mechanisms; for example, the newly introduced scope or the compensation handlers. Note that handlers are external to a scope in UML, which means that they need to be moved to the appropriate code block inside the generated BPEL scopes.

Having handled structural aspects, single actions can be converted; for example:

- *Send*: The send action is intended for sending a message to an external partner. It is modeled as a BPEL invoke with only an input variable.
- *Receive:* The receive action is intended for receiving incoming messages from external partners. It is modeled as a BPEL receive.

As an example for the transformation, Figure 3 shows the (simplified) BPEL code generated for the root scope of the UML diagram in Figure 2.

4. Related Work

Several other attempts exist to define UML extensions for service orchestrations. Most, however, require very detailed UML diagrams from designers, try to force other languages (like BPEL) on top of UML, or do not provide extensions to model vital parts of orchestrations such as compensation handling.

The work of Skogan et al. [8] has a similar focus as our approach. However, although they identify patterns to ease the transformations, the approach lacks an appropriate UML profile preventing building models at a high level of abstraction.

In a recently published article, Ermagan and Krüger [2] extend the UML with components for modeling services defining a UML profile for rich services. Collaboration and interaction diagrams are used for modeling the behavior of such components. Neither compensation nor exception handling is explicitly treated in this approach.

A first automated mapping of UML models to BPEL [1] defines a very detailed UML profile that introduces stereotypes for almost all BPEL 1.0 activities – even for those already supported in plain UML, which makes the diagrams drawn with this profile hard to read.

Another approach is shown in [4], which defines BPEL-like stereotypes to handle data flow, but does not provide support for compensation. Conversely to these approaches, UML4SOA focuses on the improvement of the expressive power of UML by defining a small set of stereotypes for modeling SOA orchestrations.

5. Conclusion and Outlook

In this paper, we have presented the UML4SOA approach for modeling service orchestrations in UML2 and utilizing these models for code generation; in particular, for generating BPEL code.

The main advantage of our approach is the provision of a concise and intuitive solution to the modeling of services in UML: a UML2 profile with a small set of model elements that allow the service engineer to produce diagrams which on the one hand visualize an orchestration of services in a easy-to-read fashion, and on the other hand contain enough information for the generation of executable code. The main aim of our transformations is the generation of comprehensible and maintainable code for further development.

We believe that being able to model service orchestrations in UML and generating executable code is an important step towards an effective model-driven development of services. We will continue to work on modeling and transformation of other service artifacts, in particular on modeling service interfaces and protocol specifications as well as investigate the need for constructs to model complex control flow patterns.

Thanks go to Alexander Knapp for fruitful discussions on the UML4SOA profile. This research has been partially supported by the EC project SENSORIA (6th Framework IST 016004).

References

[1] J. Amsden, T. Gardner, C. Griffin, S. Iyengar. "Draft UML 1.4 Profile for Automated Business Processes with a Mapping to BPEL 1.0", IBM, 2003, updated 27.12.05, ibm.com/developerworks/rational/library/content/04April/3103/3103_UMLProfileForBusinessProcesses1.1.pdf.

[2] V. Ermagan, I. Krüger. "A UML Profile for Service Modeling". In Proc. of Int. Conf. on Unified Modeling Language, LNCS 4735 Springer, 360-374, 2007.

[3] N. Koch, P. Mayer, R. Heckel, L. Gönczy, C. Montangero, "UML for Service-Oriented Systems", SENSORIA D1.4a, 2007, pst.ifi.lmu.de/projekte/Sensoria/del_24/D1.4.a.pdf.

[4] K. Mantell. "From UML to BPEL", IBM, ibm.com/developerworks/webservices/library/ws-UMLbpel/, 2005.

[5] F. Montesi, C. Guidi, G. Zavattaro. "Composing Services with Jolie". Proc. of ECOWS'07, Halle, Germany, 2007.

[6] OASIS. "Web Services Business Process Execution Language", Version 2.0 (WS-BPEL 2.0). docs.oasis-open.org/wsbpel/2.0/, visited: 01-21-08.

[7] N. Russel, A.H.M. ter Hofstede, W.M.P. van der Aalst, N. Mulyar. "Workflow Control Patterns. A Revised View", *BPM Center Report BPM-06-22*, 2006.

[8] D. Skogan, R. Grønmo, I. Solheim. "Web Service Composition in UML", Eighth IEEE International Enterprise Distributed Object Computing Conference (EDOC'04), 47-57, 2004.

[9] W3C. "Web Services Description Language", Version 1.1, www.w3.org/TR/wsdl, visited: 01-21-08.

[10] M. Wirsing, A. Clark, S. Gilmore, M. Hölzl, A. Knapp, N. Koch, A. Schroeder. "Semantic-Based Development of Service-Oriented Systems". In Proc. of FORTE06, Paris, France, LNCS 4229, pp. 24–45. Springer, 2006.