Model Driven Design of Distribution Patterns for Web Service Compositions

Ronan Barrett
School of Computing
Dublin City University
Dublin 9, Ireland
Email: rbarrett@computing.dcu.ie

Claus Pahl
School of Computing
Dublin City University
Dublin 9, Ireland
Email: cpahl@computing.dcu.ie

Abstract

Increasingly, distributed systems are being constructed by composing a number of discrete components. This practice, termed composition, is particularly prevalent within the Web service domain. Here, enterprise systems are built from many existing discrete applications, often legacy applications exposed using Web service interfaces. There are a number of architectural configurations or distribution patterns, which express how a composed system is to be deployed. However, the amount of code required to realise these distribution patterns is considerable. In this paper, we propose a novel Model Driven Architecture using UML 2.0, which takes existing Web service interfaces as its input and generates an executable Web service composition, based on a distribution pattern chosen by the software architect.

1. Introduction

The development of composite Web services is often ad-hoc and requires considerable low level coding effort for realisation [1]. This effort is increased in proportion to the number of Web services in a composition or by a requirement for the composition participants to be flexible [5]. We propose a modeling and code generation approach to address this requirement. This approach suggests Web service compositions have three modeling aspects. Two aspects, service modeling and workflow modeling, are considered by [23]. Service modeling expresses interfaces and operations while workflow modeling expresses the control and data flow from one service to another. We consider an additional aspect, distribution pattern modeling [25], which expresses how the composed system is to be deployed. Distribution patterns are an abstraction mechanism useful for modeling. Having the ability to model, and thus alter the distribution pattern, allows an enterprise to configure its systems as they evolve, and to meet varying non-functional requirements.

We base our development approach on the OMG's Model Driven Architecture (MDA) [12]. MDA considers models as formal specifications of the structure or function of a system, where the modeling language is in fact the programming language. Having rich, well specified, high level models allows for the auto-generation of a fully executable system based entirely on the model. Our models will be generated based on existing Web service interfaces, requiring only limited intervention from a software architect, who defines the distribution pattern, to complete the model.

Modeling of the composed system's distribution pattern is important, as this novel modeling aspect, provides a more complete picture of the non-functional requirements realised by a distributed system. Our approach provides a high level model which intuitively expresses, and subsequently generates, the system's distribution pattern using a UML 2.0 based Activity diagram [11]. An associated benefit of our modeling approach is the fast and flexible deployment of compositions. Motivated by these concerns, we have devised an approach, a technique and an implementation, for the model driven design of distribution patterns.

The paper is structured as follows: section two motivates distribution patterns; section three introduces our modeling and transformation approach; section four investigates our model and transformation technique, complimented by a case study; section five considers our tool implementation; section six presents related work; finally, section seven considers future work and concludes the paper.

2. Distribution Patterns

There is a subtle difference between two of the modeling aspects within a Web service composition, namely workflows and distribution patterns [25]. Both aspects refer to the high level cooperation of components, termed a collaboration, to achieve some compound novel task [22]. We consider workflows as compositional orchestrations, whereby the internal and external messages to and from services are modeled. In contrast, distribution patterns are consid-

10th IEEE International Enterprise Distributed Object Computing Conference Workshops (EDOCW'06)
0-7695-2743-4/06 $20.00 © 2006 IEEE

ered compositional choreographies, where only the external messages flow between services is modeled. Consequently the control flow between services are considered orthogonal. As such, a choreography can express how a system would be deployed. The internal workflows of these services are not modeled here, as there are many approaches to modeling the internals of such services [9, 14]. Merging both modeling efforts has been identified as future work.

Distribution patterns express how a composed system is to be assembled and subsequently deployed. These patterns are a form of platform-independent model (PIM) [12], as the patterns are not tied to any specific implementation language. The patterns identified are architectural patterns, in that they identify reusable architectural artifacts evident in software systems.

To help document the distribution patterns, discussed in this paper, a modeling notation is required. UML is a standards based graphical language for the modeling of software systems [19]. UML documents a system using two categories of diagrams, structural and behavioural.

Different distribution patterns realise different non-functional requirements. Some of these requirements are often grouped under the term, Quality of Service (QoS) [13]. In [2], four categories of QoS which affect systems at runtime are outlined, performance, dependability, safety and security. The first three categories are of particular relevance here. Some specific QoS characteristics applicable to distribution patterns, in addition to some design time issues, are detailed in [18, 21].

The patterns presented here were identified by systematically researching distribution patterns, in existing network based systems. Many of the patterns discussed here are identified by Ding et al.[10], whilst the QoS attributes of the core patterns are documented by [5, 8, 25]. However, their description in an MDA based Web service context is novel. There are three pattern categories, as follows.

- Core patterns
  - Centralised Dedicated-Hub
  - Centralised Shared-Hub
  - Decentralised Dedicated-Peer
  - Decentralised Shared-Peer
- Auxiliary patterns
  - Ring
- Complex patterns
  - Hierarchical
  - Ring + Centralised
  - Centralised + Decentralised
  - Ring + Decentralised

In order to exploit the potential of pattern-driven choreography definition, the consideration of a variety of patterns would be beneficial. To illustrate the principles, we however, focus on the two most prevalent patterns, centralised and decentralised (see Figure 1), before briefly describing the other patterns.

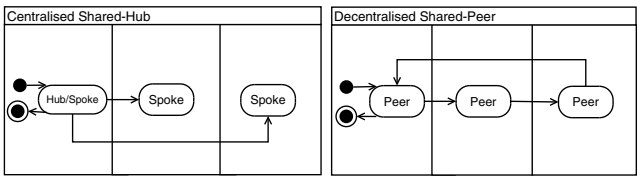

Figure 1. Examples of distribution patterns

In a centralised shared-hub pattern [5], a composition is managed in a single location by the enterprise initiating the composition. This pattern is the most widespread and is appropriate for compositions that only span a single enterprise. The advantages are ease of implementation and low deployment overhead, as only one controller is required to manage the composition. However, this pattern suffers from a communication bottleneck at the central controller. This represents a considerable scalability and availability issue for larger enterprises. The decentralised shared-peer pattern [25] addresses many of the shortcomings of the centralised shared-hub pattern by distributing the management of the composition amongst its participants. This pattern allows a composed system to span multiple enterprises while providing each enterprise with autonomy [24]. It is most important for security that each business acts upon its private data but only reveals what is necessary to be a compositional partner. In a decentralised pattern, the initiating peer is only privy to the initial input data and final output data of a composition. It is not aware of any of the intermediate participant values, unlike a centralised pattern. The disadvantages of a decentralised pattern are increased development complexity and additional deployment overheads.

We also consider two other core distribution patterns, dedicated hub and dedicated peer, which may be applied to the first two distribution patterns and their complex variants when additional autonomy, scalability and availability is required. The other distribution patterns are the ring pattern, which consists of a cluster of computational resources providing load balancing and high availability, and the hierarchical pattern, which facilitates organisations whose management structure consists of a number of levels, by providing a number of controller hubs. There are also complex variants of these distribution patterns, whereby a mix of two or more patterns are combined. Complex patterns are useful in that the combination of patterns often results in the elimination of a weakness found in a core pattern.

Two collaboration languages, Web Services Business

Process Execution Language (WS-BPEL) and Web Service Choreography Description Language (WS-CDL) [22], can enable the runtime enactment of distribution pattern based compositions. WS-BPEL, is an orchestration language whilst WS-CDL is a choreography language. The WS-CDL language provides the most obvious mapping to distribution pattern specification, as only the messages exchanged between collaborators are considered.

3. Modeling and Transformation Approach

The basis of our modeling and transformation approach is illustrated in Figure 2, as outlined by Bézivin in [6], and previously utilised in a web based engineering context by Koch in [17]. We outline the model transformation pattern from UML to our distribution pattern language, and subsequently to a collaboration language. Our relations are defined at the meta-model level using both the recently standardised QVT (Query/View/Transformation) textual and graphical notations [20]. Using QVT relations we have analysed our approach for completeness by verifying the preservation of semantics between related meta-models.

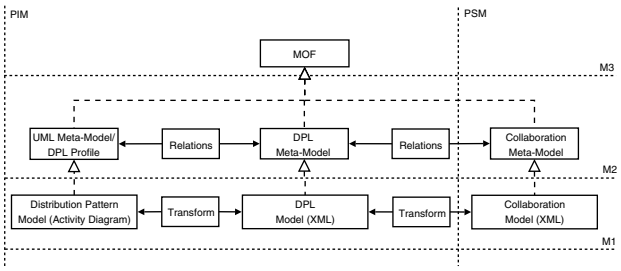

Figure 2. Model transformation pattern

An example of a QVT relation using the graphical notation can be seen in Figure 3. This relation states that the order value of a Node, from the DPL meta-model, is derived by the order value between two UML CallBahaviorAction elements, from the UML meta-model/DPL Profile.

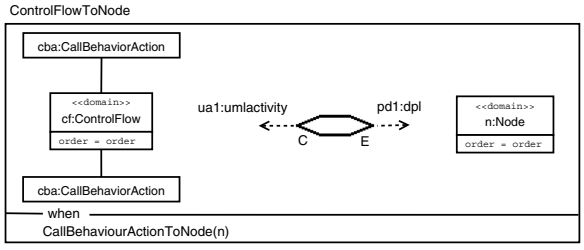

Figure 3. Example QVT relation

Our approach to distribution pattern modeling and subsequent Web service composition generation consists of five steps, as illustrated in Figure 4, and subsequently described below.

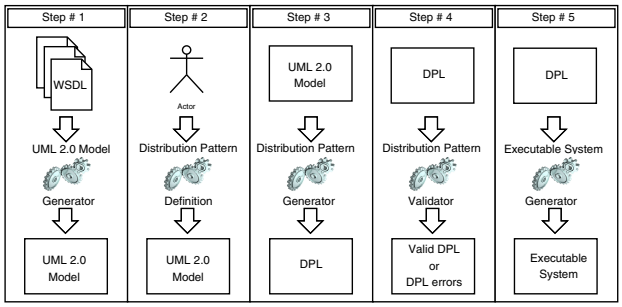

Figure 4. Overview of modeling approach

Step 1 - From Interface To Model: The initial step takes a number of Web service interfaces as input, and transforms them to the UML 2.0 modeling language standardised by OMG [11], using the UML 2.0 model generator. These interfaces represent the services which are to be composed. The model generated is based on the web services inputted, however, each service is logically separated as no composition has yet been defined.

Step 2 - Distribution Pattern Definition: The model produced in step 1 requires limited intervention from a software architect. Guided by a chosen distribution pattern, and restricted by the UML meta-model/DPL Profile (see Figure 5) , the architect must manipulate the UML model by defining connections between individual Web services and map the messages from one service to the next. Finally, the architect must set some distribution pattern specific variables on the model, which will be used to generate a distribution pattern instance. Partial automation of this step using semantics is considered in our related paper [3].

Step 3 - From Model to DPL: Using the model generated in step 2 as input, the model is transformed to a distribution pattern instance, using the distribution pattern generator. The transformation and resultant pattern instance are restricted by the DPL meta-model. This pattern instance, represented in XML using our novel specification language Distribution Pattern Language (DPL), is called a DPL document instance. The DPL specification, written in XML Schema, has no reliance on UML and so any number of modeling techniques may be used as an input. The use of this new language allows non-MOF compliant description frameworks, such as Architectural Description Languages and the π calculus, to be used in place of UML as the transformation source.

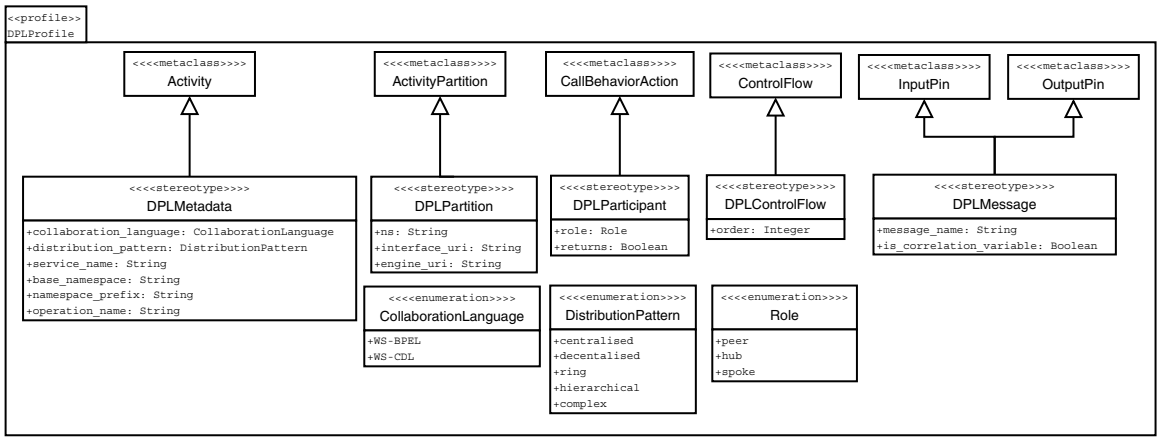

Figure 5. UML profile for modeling distribution patterns

Step 4 - Model Validation: The DPL document instance, representing the distribution pattern modeled by the software architect, is verified at this step by the distribution pattern validator, to ensure the values entered in step 2 are valid. If incorrect values have been entered, the architect must correct these values, before proceeding to the next step. Validation of the distribution pattern instance is essential to avoid the generation of an invalid system. Although this validation may be considered redundant as the pattern definition has already been restricted by the QVT relations, we envisage supporting non-QVT compliant modeling languages as set out in our future work.

Step 5 - DPL to Executable System: Finally, the executable system generator takes the validated DPL document instance and generates all the code and supporting collaboration document instances required for a fully executable system. These documents are restricted by the appropriate platform specific collaboration meta-model. This executable system will realise the Web service composition using the distribution pattern applied by the software architect. All that remains is to deploy the generated artifacts and supporting infrastructure to enable the enactment of the composed system. Dynamic deployment of the executable system is considered in our related paper [4].

4. Modeling and Transformation Technique

In this section, we introduce the techniques we have developed for the modeling and transformational approach presented in section 3, before evaluating our technique in the last subsection. There are three specific techniques listed below and elaborated in the five specific steps that follow. As before each step is illustrated in Figure 4.

- UML activity diagram/Profile extension (step 1,2)
- DPL/DPL validator (step 3,4)
- Generators (step 1,3,5)

The technique is accompanied by a small scale case study which motivates our solution. Our case study is an enterprise banking system with three interacting business processes. We choose an enterprise banking system as it is susceptible to changes in organisational structure while requiring stringent controls over data management, two important criteria when choosing a distribution pattern. The scenario involves a bank customer requesting a credit card facility. The customer applies to the bank for a credit card, the bank checks the customer's credit rating with a risk assessment agency before passing the credit rating on to a credit card agency for processing. The customer's credit card application is subsequently approved or declined.

4.1. Step 1 - From Interface To Model

As Web services' WSDL interfaces are constrained by XML Schemas, their structure is well defined. This allows us to transform the interfaces, using the UML 2.0 model generator, into a UML 2.0 activity diagram, an approach also considered by [9]. The UML model generated contains many of the new features of UML 2.0, such as Pins, CallBehaviorActions and ControlFlows.

A UML activity diagram is chosen to model the distribution pattern as it provides a number of features which assist in clearly illustrating the distribution pattern, while providing sufficient information to drive the generation of the executable system. Activity diagrams show the sequential flow of actions, which are the basic unit of behaviour, within a system and are typically used to illustrate workflows.

UML ActivityPartitions, also known as swim-lanes are used to group a number of actions within an activity diagram. In our model, these actions will represent WSDL

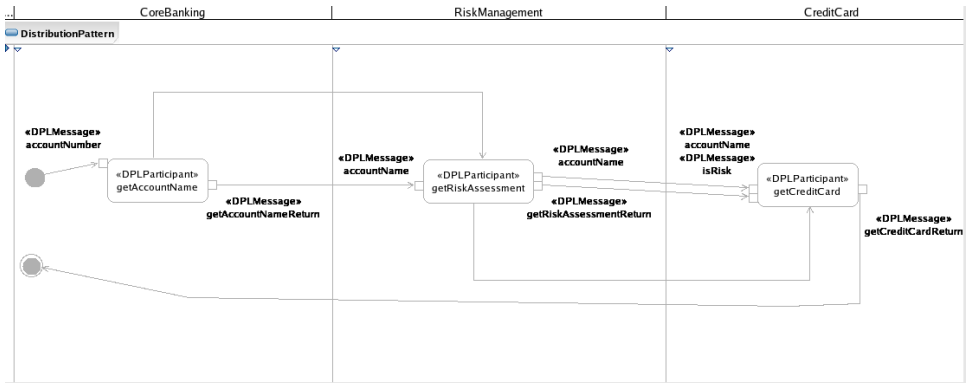

Figure 6. Generated model with connections defined by software architect, viewed in IBM RSA

operations. Any given interface has one or more ports that will have one or more operations, all of which will reside in a single swim-lane. To provide for a rich model, we use a particular type of UML action to model the operations of the WSDL interface. These actions, called CallBehaviorActions, model process invocations and have an additional modeling constructs called pins. There are two types of pins, InputPins and OutputPins, which map directly to the parts of the WSDL messages going into and out of a WSDL operation. For our UML activity diagram to effectively model distribution patterns, we require the model to be more descriptive than the standard UML dialect allows. We use a standard extension mechanism of UML, called a profile [12]. Profiles define stereotypes and subsequently tagged values that extend a number of UML constructs. Each time one of these derived constructs is used in our model we may assign values to its tagged values. An overview of our profile can be seen in Figure 5. The profile extends the Activity, ActivityPartition, CallBehaviorAction, ControlFlow, InputPin and OutputPin UML constructs. This extension allows distribution pattern metadata to be applied to the constructs via the tagged values. For example, the distribution pattern is chosen by selecting a pattern from the DistributionPattern enumeration and assigning it to the distribution_pattern tagged value on the DPLMetadata construct.

The banking case study provides three WSDL interfaces as input to the UML 2.0 model generator. These interfaces represent the bank (CoreBanking), the risk assessment agency (RiskManagement) and the credit card agency (CreditCard). All three are represented in the generated UML activity diagram, albeit without any connections between them. A swim-lane is provided for each interface. Each interface has one operation, represented as a CallBehaviorAction, which is placed in the appropriate swim-lane. The message parts associated with each operation are represented as InputPins and OutputPins. These pins are placed on the appropriate CallBehaviorAction. No model intervention from the software architect is required at this step.

4.2. Step 2 - Distribution Pattern Definition

The UML model produced in step 1, requires additional modeling. First the architect selects a distribution pattern and then assigns appropriate values to the tagged values of the stereotypes. Based on the chosen distribution pattern, the architect defines the sequence of actions by connecting CallBehaviorActions to one another, using UML ControlFlow connectors, each of which is assigned an order value. The architect then connects up the UML InputPins and OutputPins of the model, using UML ObjectFlows connectors, so data is passed through the composition.

Returning to the case study, we must connect up the three Web services to realise a distribution pattern. Before we do this, however, we select a distribution pattern appropriate to the bank's situation and requirements. The decentralised dedicated peer distribution pattern is appropriate as the bank requires credit rating information from a third party and does not wish to reveal any of the intermediate participant values of the composition. Also, the bank anticipates a high number of credit card applications, so the load must be distributed to avoid availability issues. Other scenarios would demand the use of other distribution patterns. We apply the pattern by connecting the CoreBanking and RiskManagement CallBehaviorActions together and subsequently connect the RiskManagement and CreditCard CallBehaviorActions constructs together, using ControlFlow connectors, as in Figure 6. We do not use a dedicated peer as the entry point to the composition, although this option is available to us. The InputPins and OutputPins of the CallBehaviorActions are connected together using ObjectFlow connectors, to allow the message parts propagate through the distribution pattern. An extra OutputPin, accountName, must be added to the RiskManagement CallBehaviorAction, to provide data for an InputPin, accoutName, to the CreditCard CallBehaviorAction. Finally, appropriate values must then

be assigned to the tagged values of the stereotypes.

4.3. Step 3 - From Model to DPL

The UML model completed in step 2 may now be transformed to a DPL document instance by the distribution pattern generator. This document, which is at the same level of abstraction as the UML model, is an internal representation of the distribution pattern which can be validated. The DPL specification, written in XML Schema, and the document instance, an XML file, have no reliance on UML and so provide for interoperability with other modeling techniques. Figure 7 shows the DPL document instance for the case study. The message names and message parts have been truncated for space reasons.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<dpl:pattern_definition xmlns:dpl="http://localhost/dpl"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://localhost/dpl dpl.xsd">
  <dpl:collaboration_language>WS-BPEL</dpl:collaboration-language>
  <dpl:distribution_pattern>decentralised</dpl:distribution-pattern>
  <dpl:service_name>BankingPeerToPeer</dpl:service-name>
  <dpl:base_namespace>BankingPeerToPeer</dpl:base-namespace>
<dpl:namespace_prefix>http://foo.com/wsdl/</dpl:namespace_prefix>
  <dpl:operation_name>applyForCC</dpl:operation-name>
  <dpl:correlation_variables>
    <dpl:variable name="accountNumber" type="xsd:int"/>
  </dpl:correlation_variables>
  <dpl:nodes>
   <dpl:node returns="true" name="CoreBanking" ns="http://CoreBanking"
     uri="http://local/CB?WSDL" euri="http://local:1234/" order="1" role="peer"/>
   <dpl:node name="RiskManagement" ns="http://RiskManagement"
     uri="http://local/RM?WSDL" euri="http://local:1234/" order="2" role="peer">
    <dpl:mappings>
     <dpl:mapping>
      <dpl:from message="getANResponse" part="getANReturn" node="CoreBanking"/>
       <dpl:to message="getRARequest" part="accountName" node="RiskManagement"/>
      </dpl:mapping>
     </dpl:mappings>
   </dpl:node>
   <dpl:node name="CreditCard" ns="http://CreditCard"
     uri="http://local/CC?WSDL" euri="http://local:1234/" order="3" role="peer">
    <dpl:mappings>
     <dpl:mapping>
      <dpl:from message="getRARequest" part="accountName" node="CoreBanking"/>
      <dpl:to message="getCCRequest" part="accountName" node="CreditCard"/>
     </dpl:mapping>
     <dpl:mapping>
      <dpl:from message="getRAResponse" part="getRAReturn" node="RiskManagement"/>
      <dpl:to message="getCCRequest" part="isRisk" node="CreditCard"/>
     </dpl:mapping>
    </dpl:mappings>
   </dpl:node>
  </dpl:nodes>
</dpl:pattern-definition>
```

Figure 7. DPL document instance

With regard to our case study, many of the values in Figure 7 are the same as the values applied by the software architect in step 2, such as distribution_pattern and service_name. The ControlFlow connectors previously defined between the CallBehaviorActions are used to assign an order value to the dpl:nodes, which themselves are derived from the CallBehaviorActions (getAccountName, getRiskAssessment and getCreditCard) in the UML model. The ObjectFlow connectors between the InputPins and OutputPins are used to define the mappings between dpl:nodes. The first dpl:node does not require any explicit ObjectFlow connectors as the initial values passed into the system are used as its input automatically.

4.4. Step 4 - Model Validation

To verify the model, the DPL document instance is verified against the DPL Schema, by the distribution pattern validator. The verification process ensures the distribution pattern selected by the software architect is compatible with the model settings. For example, in our case study, as the decentralised dedicated-hub distribution pattern has been chosen, there must be at least two dpl:nodes having a peer role and there must not be any dpl:nodes with a hub role. If any errors are detected they must be corrected by the software architect by returning to step 2.

4.5. Step 5 - DPL to Executable System

The verified DPL document instance is now used by the executable system generator to generate all the interaction logic documents and interfaces required to realise the distribution pattern. The generator creates interaction logic documents based on the collaboration-language setting. Additional WSDL interfaces are also generated, if necessary. The system is now executable and ready for deployment.

In our case study example, three WS-BPEL interaction logic documents are created to represent each of the three peers in the distribution pattern. Additionally, three WSDL interfaces are created as wrappers to each interaction logic document, enabling the composition to work in a decentralised environment.

4.6. Evaluation

To assess our approach, we use the criteria set out in [23], along with some of our own success criteria.

- Pattern expression - We have identified a number of distribution patterns and have shown how patterns can be expressed sufficiently using UML with our DPL-Profile extension and in XML, using DPL.
- Verification - We have verified our model transformations using QVT relations between corresponding meta-models.
- Readability - Our modeling approach, which visualises the distribution pattern, should be intelligible to software architects. As the model is at the PIM level, clutter from implementation details is avoided.
- Executable - Our UML model and associated profile is sufficiently rich to generate a DPL document instance and subsequently all the interaction logic and interface documents needed to create an executable system.
- Maintenance overhead - Our MDA approach allows easy manipulation of the system's distribution pattern.

5. Tool Implementation

TOPMAN (TOPology MANager) is our solution to enabling distribution pattern modeling using UML 2.0 and subsequent Web service composition generation. The only technologies required by the tool are the Java runtime and both an XML and XSLT parser. The tool implementation is illustrated in Figure 8.

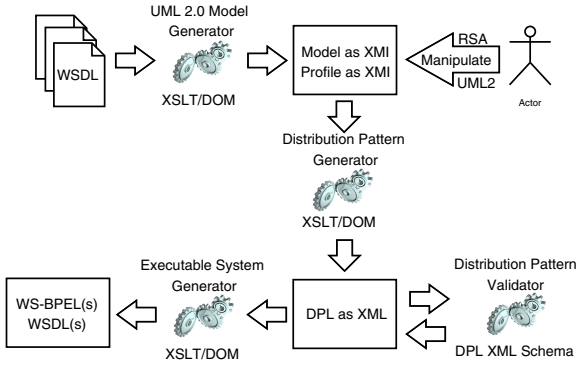

Figure 8. Overview of TOPMAN tool

The UML 2.0 model generator uses XSLT to transform the WSDL interfaces of the Web services participants, to a UML 2.0 activity diagram, which generates, using XML DOM, an XMI 2.0 [15] document. XMI is the XML serialisation format for UML models. The model generated includes a reference to our predefined UML profile for distribution patterns, which is also serialised to XMI 2.0.

A number of tools may be used to describe the distribution pattern. IBM's commercial tool Rational Software Architect (RSA) is compatible with XMI 2.0 and supports many of the UML 2.0 features. The tool has a GUI which allows the software architect to define the distribution pattern. Upon completion, the model can be exported back to XMI for further processing by TOPMAN. An alternative to IBM's commercial tool is UML2, an open source tool supporting UML 2.0, which allows the model to be viewed and manipulated in an editor. Unfortunately, there are currently no open source GUI based UML tools which support exporting an XMI 2.0 representation of a UML 2.0 model.

The distribution pattern generator uses XSLT to transform the UML 2.0 model to a DPL instance document. The DPL document instance is then verified by an XML validating parser. Finally the DPL document instance is used to drive the executable system generator, resulting in the creation of an executable composition. Within the executable system generator, XSLT and XML DOM are used to generate the interaction logic and interface documents needed by a workflow engine to realise the distribution pattern. Each transformation is written to implement a previously defined QVT relation between source and target meta-models. Ideally, a choreography based specification, such as WS-CDL, should be used. However, there is no enactment engine currently available for WS-CDL. Instead, we choose to use an open source WS-BPEL engine, activeBPEL. Although WS-BPEL is an orchestration engine, we can use it to apply distribution patterns based on the work in [8].

6. Related Work

Two workflow management systems motivate and provide concrete implementations for two of the distribution patterns explored in this paper. However, neither system provides a standards-based modeling solution to drive the realisation of the chosen distribution pattern. The first system DECS [25], from which the distribution pattern term originates, is a workflow management system, which supports both centralised and decentralised distribution patterns, albeit without any code generation element. DECS defines elementary services as tasks whose execution is managed by a coordinator at the same location. The second system SELF-SERV [24], proposes a declarative language for composing services based on UML 1.x statecharts. SELF-SERV provides an environment for visually creating a UML statechart which can subsequently drive the generation of a proprietary XML routing table document. Pre- and post-conditions for successful service execution are generated based on the statechart inputs and outputs. The authors' more recent work [16] considers the conformance of services with a given conversational specification using a more complete model-driven approach. A mapping from SELF-SERV to WS-BPEL is also investigated.

From the modeling perspective Grønmo et al. [23, 9], consider the modeling and building of compositions from existing Web services using MDA, an approach similar to ours. However, they consider only two modeling aspects, service (interface and operations) and workflow models (control and data flow concerns). The system's distribution pattern is not modeled, resulting in a fixed centralised distribution pattern for all compositions. Their modeling effort begins with the transformation of WSDL documents to UML, followed by the creation of a workflow engine-independent UML 1.4 activity diagram (PIM), which drives the generation of an executable composition. Additional information required to aid the generation of the executable composition is applied to the model using UML profiles.

Another approach of interest is an extension of WebML, which uses the Business Process Modeling Notation (BPMN), instead of UML, for describing Web service processes [7]. The authors consider the assignment of processes to servers, termed process distribution. However, the approach is at a lower conceptual level than that of distribution patterns as communication modes between services are explicitly modeled.

7. Conclusion and Future Work

An engineering approach to the composition of service-based software systems is required. We have introduced techniques based on architectural modeling and pattern-based development, which have already been applied successfully in both object-oriented and component-based systems. We have also applied patterns, which have been found useful in a networking context, to the Web service domain. Our contribution is a modeling and transformation approach, technique and implementation for expressing the distribution pattern of a Web service composition. Our novel modeling aspect, distribution patterns, expresses how a composed system is to be deployed, providing for improved maintainability and comprehensibility. Any of the distribution patterns discussed may be used to guide the generation of an executable system, based on the enterprises requirements. Three modeling and transformation techniques were introduced, along with a tool (TOPMAN) which assists in the generation of an executable system guided by the chosen pattern.

We intend considering alternatives to our UML modeling language approach, based on π calculus and Architecture Description Languages. In addition, quantitative analysis of the reduction in coding effort due to our modeling approach would provide additional motivation for our work. Finally, documentation of the QoS attributes of the complex patterns is also an important future effort.

8. Acknowledgment

The authors would like to thank the Irish Research Council for Science, Engineering and Technology IRCSET.

References

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architecture and Applications*. Springer Verlag, 2004.

[2] M. Barbacci. Quality Attributes. Technical report, CMU/SEI-95-TR-021, 1995.

[3] R. Barrett and C. Pahl. Semi-Automatic Distribution Pattern Modeling of Web Service Compositions using Semantics. In *Proc. Tenth IEEE International EDOC Conference (to appear)*, Hong Kong, China, October 2006.

[4] R. Barrett, C. Pahl, L. Patcas, and J. Murphy. Model Driven Distribution Pattern Design for Dynamic Web Service Compositions. In *Proc. Sixth International Conference on Web Engineering (to appear)*, Palo Alto, USA, July 2006.

[5] B. Benatallah, Q. Z. Sheng, and M. Dumas. The self-serv environment for web services composition. *IEEE Internet Computing*, 7:40–48, 2003.

[6] J. Bézivin. In search of a basic principle for model driven engineering. *UPGRADE*, 2, 2004.

[7] M. Brambilla, S. Ceri, P. Fraternali, and I. Manolescu. Process modeling in web applications. *ACM Transactions on Software Engineering and Methodology (to appear)*.

[8] G. B. Chafle, S. Chandra, V. Mann, and M. G. Nanda. Decentralized orchestration of composite web services. In *Proc. 13th international World Wide Web conference*, pages 134 – 143, New York, NY, USA, May 2004.

[9] D. Skogan and R. Grønmo and I. Solheim. Web Service Composition in UML. In *Proc. 8th International IEEE Enterprise Distributed Object Computing Conference (EDOC)*, pages 47–57, Monterey, California, September 2004.

[10] C. H. Ding, S. Nutanong, and R. Buyya. P2P Networks for Content Sharing. *CoRR*, cs.DC/0402018, 2004.

[11] H. E. Eriksson, M. Penker, B. Lyons, and D. Fado. *UML 2 Toolkit*. Wiley, 2003.

[12] D. S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, 2004.

[13] S. Frolund and J. Koistinen. Quality of service specification in distributed object systems design. In *Proceedings of the 4th USENIX Conference on ObjectOriented Technologies and Systems (COOTS)*, New Mexico, USA, April 1998.

[14] T. Gardner. Uml modeling of automated business processes with a mapping to bpel4ws. In *Proc. First European Workshop on Object Orientation and Web Service (EOOWS)*, Darmstadt, Germany, July 2003.

[15] T. J. Grose. *Mastering XMI: Java Programming with XMI, XML, and UML*. Wiley, 2002.

[16] K. Baïna and B. Benatallah and F. Casati and F. Toumani. Model-driven web service development. In *Proc. 16th International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 290–306, Riga, Latvia, June 2004.

[17] N. Koch. Transformations Techniques in the Model-Driven Development Process of UWE. In *Proc. of 2nd Model-Driven Web Engineering Workshop (to appear)*, Palo Alto, USA, July 2006.

[18] D. A. Menascé. QoS Issues in Web Services. *IEEE Internet Computing*, 6(6):72–75, November–December 2002.

[19] OMG. Unified Modeling Language (UML), version 2.0. Technical report, OMG, 2003.

[20] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Final Adopted Specification. Technical report, OMG, 2005.

[21] OMG. UML Profile for QoS and Fault Tolerance. Technical report, OMG, 2005.

[22] C. Peltz. Web services orchestration and choreography. *IEEE Computer*, 36, 2003.

[23] R. Grønmo and I. Solheim. Towards modeling web service composition in uml. In *Proc. 2nd International Workshop on Web Services: Modeling, Architecture and Infrastructure (WSMAI-2004)*, pages 72–86, Porto, Portugal, April 2004.

[24] Q. Z. Sheng, B. Benatallah, and M. Dumas. Self-serv: A platform for rapid composition of web services in a peer-to-peer environment. In *Proc. 28th International Conference on Very Large Data Bases*, pages 1051–1054, Hong Kong, China, August 2002.

[25] S. J. Woodman, D. J. Palmer, S. K. Shrivastava, and S. M. Wheater. A system for distributed enactment of composite web services. In *Work in progress report, Int. Conf. on Service Oriented Computing*, Trento, Italy, December 2003.