Distribution Pattern-driven Development of Service Architectures

Ronan Barrett and Claus Pahl
(Dublin City University, Ireland
(rbarrett|cpahl@computing.dcu.ie)

Abstract: Distributed systems are being constructed by composing a number of discrete components. This practice is particularly prevalent within the Web service domain in the form of service process orchestration and choreography. Often, enterprise systems are built from many existing discrete applications such as 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 in a distributed environment. However, the amount of code required to realise these distribution patterns is considerable. In this paper, we propose a distribution pattern-driven approach to service composition and architecting. We develop, based on a catalog of patterns, a UML-compliant framework, which takes existing Web service interfaces as its input and generates executable Web service compositions based on a distribution pattern chosen by the software architect.

Key Words: Service-oriented architecture, service composition, distribution pattern, architecture modelling, service process generation.
Category: C.2.4 - Distributed Systems, D.2.2 - Design Tools and Techniques, D.2.7 - Distribution, Maintenance, and Enhancement, D.2.11 - Software Architectures

1 Introduction

Distributed systems are being constructed by composing a number of discrete components. This practice is particularly prevalent within the Web service domain in the form of service process orchestration and choreography, see e.g. [1, 43]. Often, enterprise systems are built from many existing discrete applications such as legacy applications exposed as Web services. The services are glued together through their interfaces, realising a service-oriented architecture.

The development of composite Web services is often ad-hoc and requires considerable low level coding for realisation [1]. This effort increases with the number of Web services in a composition or by a requirement for the composition participants to be flexible [12]. We propose to focus on architecture modelling to address these requirements. We look at Web service compositions from three architectural perspectives:

- Service modelling expresses service functionality in terms of interfaces and operations.
- Workflow and process composition expresses the control and data flow from one service to another.
- Distribution pattern-based architecting expresses how the composed system is to be deployed.

While aspects of the first two have been considered in [39], the crucial distribution topology of a distributed service architecture is often neglected – although critical quality factors depend on it. Distribution patterns are an abstraction mechanism useful for architecture modelling [44]. 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. Distribution patterns reflect for instance the centralised or decentralised nature of these systems. Patterns express proven techniques, which make it easier to reuse successful designs and architectures [22].

Having detailed high-level models also allows for the generation of a fully executable system based entirely on the model [34]. Our architecture models consist of services as building blocks and compositions based on distribution patterns. Executable service processes can be generated based on the service distribution architecture, requiring only limited intervention from a software architect, who determines the distribution pattern to define and configure a model. Modelling of the composed system's distribution pattern is important, as this currently often neglected modelling aspect provides insight into the non-functional properties realised by a distributed system. Emphasising distribution patterns allows architecture design to be driven by quality considerations.

Motivated by these concerns, we have developed languages and techniques for the distribution pattern-driven modelling of compositions in service architectures. An associated benefit of our modelling approach is the fast and flexible deployment of compositions. Our approach provides a high-level model which intuitively expresses, and subsequently generates, the system's distribution pattern using a UML2-based activity diagram [18]. We develop, based on a catalog of patterns, a UML-compliant framework, which takes existing Web service interfaces as its input and generates executable Web service compositions based on a distribution pattern chosen by the software architect. A central constraint of distribution pattern-driven architecture is that distribution is only one aspect in the architecture of a complex system. Consequently, integration and interoperability of the notations and techniques presented here with other techniques and languages is a prerequisite for the effectiveness of the approach. We therefore address integration and interoperability in particular.

The paper is structured as follows: Section 2 introduces service architecture principles, motivates distribution patterns in the context of service architectures and outlines a pattern catalog; Section 3 introduces our modelling and transformation approach. In Section 4, we apply the techniques in the context of a banking sector case study and discuss some quality implications. In Section 5 we evaluate the approach; Section 6 presents related work; and, finally, Section 7 concludes the paper.

2 Service Architectures and Distribution Patterns

2.1 Service Architectures

Services, particularly Web services, are building blocks of compositions in distributed environments. Services are composed to processes, executed by suitable process engines. There is an important difference between two of the modelling aspects within a process-oriented service composition, namely workflows and distribution patterns [44]. Both aspects refer to the high-level cooperation of components, termed a collaboration, to achieve some compound task [37].

- We consider workflows as compositional orchestrations, where the internal and external messages to and from services are modelled.
- In contrast, distribution patterns are considered as compositional choreographies, where only the external messages flow between services is modelled.

Consequently, the external control flow between services can be considered orthogonal to the internal orchestration perspective. As such, a choreography can express how a system would be deployed. The internal workflows of these services shall be neglected here, as there are many approaches to modelling the internals of such services [15, 21]. Distribution includes additionally aspects in relation to the location and role of participants in a composition. In particular this interaction of different roles in a distributed, networked environment determines crucial non-functional properties.

2.2 Distribution Patterns

Distribution patterns are frequently used solutions that express how a composed system is to be assembled and subsequently deployed in terms of its distribution topology. These patterns are platform-independent [20], as the patterns are not tied to any specific implementation language. The patterns identified are architectural patterns, in that they define reusable architectural artifacts evident in software systems [8].

To help document the distribution patterns, a modelling notation is required. UML is a standardised graphical language for the modelling of software systems [28]. UML documents a system using two categories of diagrams, structural and behavioural. Structural diagrams describe the static structures of a system along with the interrelations between components of the system. Behavioural diagrams describe the dynamic behaviour of a system. Structural diagrams such as UML class diagrams are appropriate for service modelling, while UML activity diagrams or interaction diagrams are appropriate for modelling both workflows and distribution patterns. While interaction diagram seem initially more intuitive in the context of the choreographic composition, we use activity diagrams that have been extended recently by connectivity features (such as pins and control flow elements that link to Web service technologies). Activity diagrams more naturally suited to capture process composition.

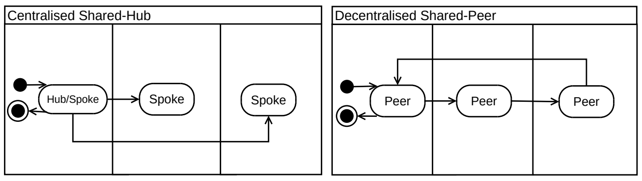

Figure 1: Examples of distribution patterns

2.3 Pattern Catalog

In order to exploit the potential of pattern-driven service composition, the consideration of a variety of patterns is beneficial. Distribution patterns need to convey the following aspects:

– structural connectivity of services as components,

– message interaction direction (internal perspective),

– activation/initiation and response (external perspective).

We will now introduce a distribution pattern catalog. The patterns presented here were identified by systematically researching distribution patterns in existing network-based systems. Many of the patterns discussed here are identified in [16]. We applied the extracted patterns to five different application architecture designs – details are presented in Section 6.2. We have combined the different findings in the context of service architectures based on our own review [5].

We have identified seven patterns in three pattern categories, as follows:

– Core patterns

  - Centralised Hub: Manages the composition from a single location, normally the participant initiating the composition. The composition controller (the hub) is located externally from the service participants to be composed (the spokes). Two messages are exchanged between the hub and a spoke for each spoke execution i.e. synchronous communication. The composition completes after the final spoke has completed execution and has returned a response to the hub. This is the most popular, default distribution pattern configuration for compositions.

  - Decentralised Peer: Distributes the management of the composition amongst the composition. The participant which initiates the composition is located externally from the other participants. Only one message is exchanged between the caller and the callee for each peer execution (asynchronous communication). The composition completes after the final peer has completed execution and has returned execution control to the peer which commenced the composition.

These are illustrated in Figure 1.

– Auxiliary patterns

  - Ring: Identical participants acting as a cluster. The pattern by itself does not facilitate composition and is normally used in association with other patterns. There are no start and end points to the ring pattern. The Ring pattern provides fault-tolerant infrastructure to a Web Service composition. The specific ring implementation defines, at the mirror head, the algorithm for determining how the load is delegated amongst the ring participants.

– Complex patterns

  - Hierarchical: Tree-based structure featuring a number of controller hubs. The pattern is related to the Centralised pattern. Two messages are exchanged between the hub and a spoke for each spoke execution (synchronous communication). Two messages are also exchanged whenever hubs intercommunicate. The composition completes after the final spoke has completed execution and has returned a response to its hub, which then returns control to its owning hub, until finally the parent hub regains control of the composition and terminates.

  - Ring + Centralised: The Ring + Centralised pattern combines the Ring pattern with the Centralised pattern. This complex pattern eliminates the single point of failure and communication bottleneck at the central hub by providing a number of identical redundant hubs organised as a ring. As with the core centralised pattern, messages are exchanged between the hub and a spoke for each spoke execution (synchronous communication). The composition completes after the final spoke has completed execution and has returned a response to the hub. The specific ring implementation defines the algorithm for determining how the load is delegated amongst the ring participants.

  - Centralised + Decentralised: The Centralised + Decentralised pattern combines the Centralised pattern with the Decentralised pattern. This complex pattern allows a number of participants to function as hubs locally whilst functioning as peers within the larger composition. Only one message is exchanged between each hub/peer for each execution (asynchronous communication). The composition completes after the final hub/peer has completed execution and has returned execution control to the hub/peer which commenced the composition. Two messages are exchanged between the hub/peer and a spoke for each spoke execution (synchronous communication).

  - Ring + Decentralised: The Ring + Decentralised pattern combines the Ring pattern with the Decentralised pattern. This complex pattern uses one or more rings to provide redundant copies of participants. As with the core decentralised pattern, only one messages is exchanged between the caller and the callee for each peer execution (asynchronous communication). The composition completes after the final peer has completed execution and has returned execution control to the peer which commenced the composition. The specific ring implementation defines the algorithm for determining how the load is delegated amongst the ring participants. An example of this pattern in an existing network context, is a file sharing system whose peers have load balancing enabled.

Core patterns capture the principle forms of distribution that can commonly be observed in service compositions. Auxiliary patterns are patterns which can be combined with core patterns to alter a given non-functional quality attribute of a core pattern. Complex patterns can be formed by combining core patterns – we have presented a selection of common ones.

3 Architecture Modelling and Transformation

Our approach to distribution pattern modelling and subsequent Web service composition generation consists of a number of modelling, validation and generation activities, as illustrated in Figure 2. We will introduce and discuss the three activities in the subsequent subsections.

3.1 Modelling and Transformation Languages

We introduce the modelling and transformational techniques we have developed for the distribution pattern-driven service architecture approach. There are three specific techniques:

– Modelling: Pattern-driven service architecture modelling based on a UML activity diagram extension using the profile mechanism.

– Validation: Internal representation and analysis of architecture and model correctness using the distribution pattern language DPL.

– Generation: Generation of composite executable service processes from the architecture models.

The basis of our modelling and transformation integration and interoperability approach (Fig. 3) is outlined by Bézivin in [3], and previously utilised in a Web-based engineering context by Koch in [26]. MOF plays the central role in defining a range of notations and integrating them into a conversion and transformation framework. We outline the model transformation pattern from UML to our distribution pattern language DPL, and subsequently to a collaboration language. Relations are abstract specifications

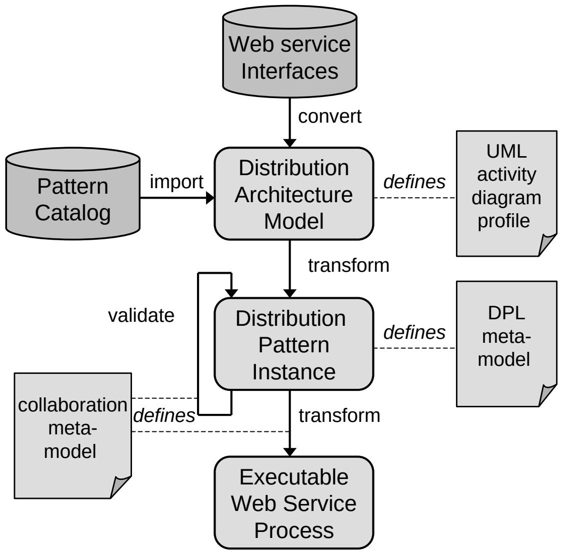

Figure 2: Overview of modelling approach

of transformations (conversion between UML and DPL and generation of executables from DPL) and transforms are their actual implementation [9]. Is it important here to distinguish the different purposes of the UML profile and DPL. The UML profile enhances UML activity diagrams with distribution-specific aspects. It acts as an interface language. DPL is an internal, intermediate language that captures the same in formation as the UML extension. However, its purpose is to allow more interface languages than UML to be dealt with and it also act as the basis for internal processing like analysis and transformation.

Although our modelling solution is independent of the actual execution language, it is important to discuss (executable) collaboration languages to fully understand how a modelled system will interact at run time. Ultimately, the patterns will be realised at this level. Two collaboration languages, Web Service Business Process Execution Language (WS-BPEL) and Web Service Choreography Description Language (WS-CDL) [37], 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 link to distribution pattern specification as only the messages exchanged between collaborators are considered. However, the authors of [13] argue that both languages are appropriate target languages for our context. It is interesting to note that the use of WS-BPEL in a centralised distribution pattern is considered inevitable by the authors of [44]. A WS-BPEL engine may be used in a peer-to-peer distribution pattern, which improves throughput and scalability and results

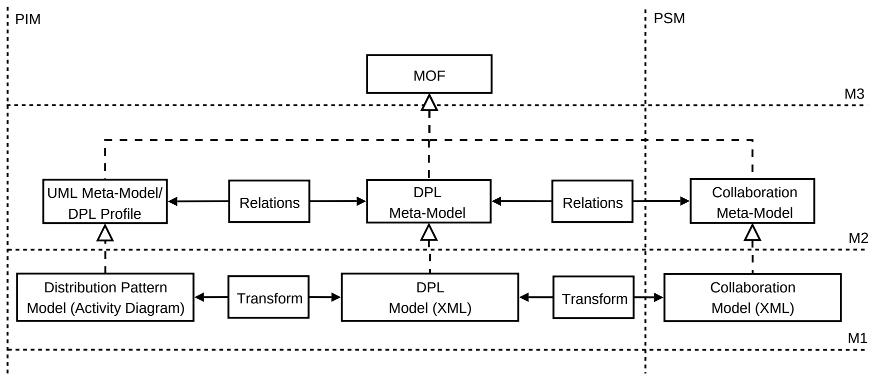

Figure 3: Modelling and transformation languages

in lower response times when compared to the same engine in a centralised pattern.

3.2 Modelling of Distribution Architectures

In order to be compatible with other modelling notations, we develop a UML-based graphical modelling notation for distribution pattern-driven service composition. We introduce the notation and the architecture modelling activities now.

3.2.1 UML-based Distribution Pattern Modelling

The first architecture modelling activity takes a number of Web service interfaces in the form of WSDL specifications as input, and transforms them to the UML 2.0 modelling language, standardised by OMG [18], using a UML 2.0 model generator we developed for our environment. These interfaces represent the services which are to be composed. The model generated is an initially unconnected collection of Web services interfaces, i.e. each service is logically separated as no composition has yet been defined.

As WSDL interfaces are constrained by XML schemas, their structure is well defined. This allows us to transform the interfaces into a UML 2.0 activity diagram – an approach also considered by [15]. The UML model generated contains many of the 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 visualising 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. We define our pattern architecture notation in terms of MOF M2 metamodels (see Fig. 3) in two steps:

– Use of existing UML activity diagram elements to model principles of service distribution.

– Extension of UML activity diagrams using the profile mechanism to cater for specific distribution aspects.

We start with an explanation of the central UML activity diagram metamodel elements that we utilise to model distribution in service architectures:

– UML ActivityPartitions, also known as swim-lanes, are used to group a number of actions within an activity diagram. In our model, these actions represent WSDL operations. Any given interface has one or more ports that has one or more operations, all of which 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.

– These actions are enhanced by an additional modelling construct 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 models 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 [20], to enhance the metamodel. Profiles define stereotypes and subsequently tagged values that extend 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 Fig. 4. 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.

3.2.2 Distribution Pattern Definition

The initial UML-based services model is the basis for the software architect to select a distribution pattern and apply an instance of that pattern to the UML model.

– Pattern Instantiation: First, the architect selects a distribution pattern and then assigns appropriate values to the tagged values of the stereotypes to define the pattern instance. Guided by a chosen distribution pattern and restricted by the UML metamodel/DPL Profile (see Figure 4), the architect must determine the service distribution architecture model by instantiating connections between individual pattern instance elements as services and map the messages from one service to the next. Based on the chosen pattern, the architect defines the sequence of actions by connecting CallBehaviorActions to one another, using UML ControlFlow connectors,

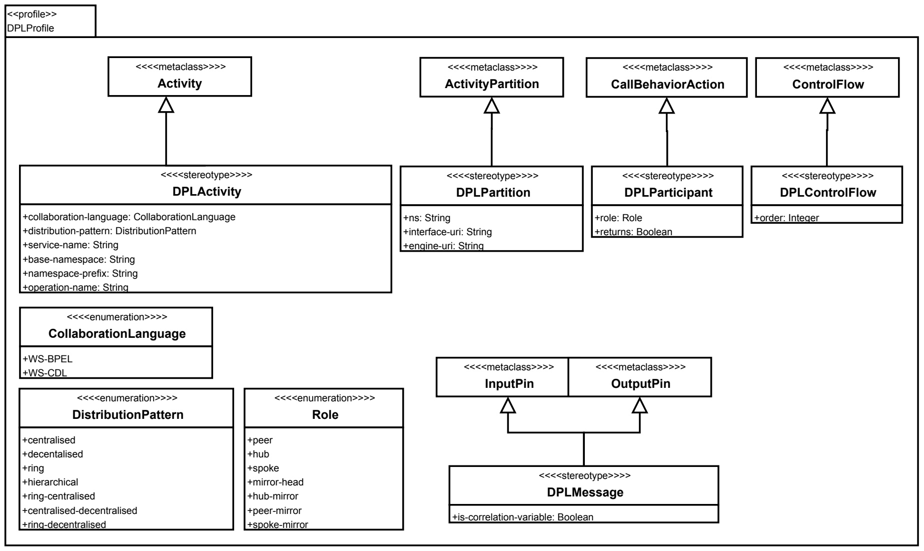
<<profile>>
DPLProfile
<<<<metaclass>>>> Activity
<<<<metaclass>>>> ActivityPartition
<<<<metaclass>>>> CallBehaviorAction
<<<<metaclass>>>> ControlFlow
<<<<stereotype>>>> DPLActivity
+collaboration-language: CollaborationLanguage
+distribution-pattern: DistributionPattern
+service-name: String
+base-namespace: String
+namespace-prefix: String
+operation-name: String
<<<<stereotype>>>> DPLPartition
+ns: String
+interface-uri: String
+engine-uri: String
<<<<stereotype>>>> DPLParticipant
+role: Role
+returns: Boolean
<<<<stereotype>>>> DPLControlFlow
+order: Integer
<<<<enumeration>>>> CollaborationLanguage
+WS-BPEL
+WS-CDL
<<<<enumeration>>>> DistributionPattern
+centralised
+decentalised
+ring
+hierarchical
+ring-centralised
+centralised-decentralised
+ring-decentralised
<<<<enumeration>>>> Role
+peer
+hub
+spoke
+mirror-head
+hub-mirror
+peer-mirror
+spoke-mirror
<<<<metaclass>>>> InputPin
<<<<metaclass>>>> OutputPin
<<<<stereotype>>>> DPLMessage
+is-correlation-variable: Boolean

Figure 4: UML profile for modelling distribution patterns

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.

– Generation Customisation: The architect can initialise some distribution pattern-specific variables on the model, which will be used to generate a distribution pattern instance. This in turn provides the necessary details to generate executable service compositions. The partial automation of this step using semantics is considered in our previous work [10].

3.3 Validation of Distributed Pattern Architectures

Before an executable service process is generated, the service architecture model is transformed into an abstract intermediate representation – the Distribution Pattern Language (DPL). Its purpose it to provide an internal notation – independent from interface languages such as UML, thus allowing different input languages to be used. Analysis and validation can be carried out before executables are generated.

3.3.1 Intermediate Representation – Distribution Pattern Language

The pattern-based service distribution 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 document, which is at the same level of abstraction as the UML model, is an internal representation of the distribution pattern instance which can be validated. This pattern instance, represented in XML using our specification language Distribution Pattern Language (DPL), is called a DPL document instance.

We follow a two-pronged semantic approach for both languages and transformations here:

– A definition in terms of MOF to obtain interoperable languages and transformations, suitable for platforms such as UML, EMF, ECore, XML, XMI, in order to guarantee interoperability.

– Formal definitions of languages and transformations based on graph theory are used to formally integrate and provide a basis to analyse the models and transformations.

In this paper, as stated above, we will focus on the first aspect. However, we briefly outline the principles of the formal semantics. The DPL semantics are defined in terms of attributed, typed directed graphs – a common formal framework for process-oriented compositions [42, 17, 7]:

– Each node represents a service with its type, which reflects the role the service plays in a distribution topology.

– Attributes capture the meta-level stereotype attributes.

– Edges represent the service connectivity.

These three graph elements are directly reflected by the three sections of a DPL instance. As [42, 7] demonstrate, this approach is also suitable to formally define UML activity diagrams and, therefore, allows us to verify the correctness of the mapping between UML and DPL.

The DPL specification, written in XML Schema, has no reliance on UML and so any number of modelling 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 formal specification language suitable for distribution modelling such as the π-calculus to be used in place of UML as the transformation source [32].

3.3.2 Distribution Architecture Validation

The DPL document instance, representing the distribution pattern modelled by the software architect, is verified at this stage by the distribution pattern validator to ensure the values entered as attributes are valid. If incorrect values have been entered, the architect must correct these values before proceeding to the generation stage. Validation of the distribution pattern instance is essential to avoid the generation of an invalid system.

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.

Validating attributes is one possible form of validation. At the DPL-level, the following analyses are in general possible:

- Attribute consistency (as explained above).
- Structural pattern graph consistency (such as detecting incorrect cycles in centralised architectures or identification of isolated services).
- Structural semantic consistency (such as the correct association of node types corresponding to the graph structure-implied role).

A suitably constrained architecture model editor can already detect the latter two categories. Nonetheless, these two are important if models are imported and a previous validation cannot be relied on. The pattern definition could also be restricted by the QVT relations, which we use and which would make some validation considered redundant. However, we envisage supporting non-QVT compliant modelling languages in the future.

3.4 Generation of Executable Service Processes

Finally, the executable system generator takes the validated DPL document instance and generates the code and supporting collaboration document instances required for a fully executable system. These documents are restricted by the appropriate collaboration meta-model. This executable system realises 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. Additional WSDL interfaces are also generated, if necessary. Dynamic deployment of the executable system is considered in more detail in [11]. We only introduce the transformation technique here.

Our relations are defined at the meta-model level using the recently standardised QVT (Query/View/Transformation) graphical notation [29]. An example of a QVT relation can be seen in Fig. 5. OperationToVariable defines the relation between a DPL Operation artifact and BPEL Variable artifacts. The relation is called by another relation, the PatternDefinitionToProcess. We have implemented the transformation using ATL, a hybrid model transformation language that is well-supported by tools. QVT has acted as an abstract specification for the ATL transformation implementation. The following ATL specification implements the QVT definition from Fig. 5:

```
rule OperationToVariable ( type : String, op : DPL!Operation ) {
  to
    var BPEL!Variable( name <- op.eContainer().eContainer().name +
                       op.name + type.messageType <- mes ),
    mes : BPEL!Message( qName <- op.eContainer().eContainer().ns +
```

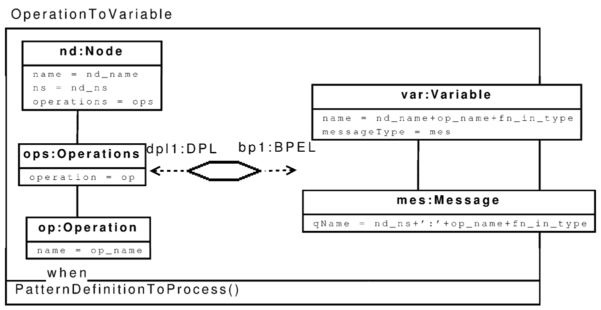

Figure 5: QVT specification – Operation To Variable

```
                          ':'op.name+type )
    do{var:}
}
```

Using QVT and the underlying graph-theoretic foundations, which define the transformations formally [40], we have analysed our approach for completeness by verifying the preservation of semantics between related meta-models.

A collaboration language like WS-BPEL is the central target language. However, a Deployment Catalog notation complements the generation process. While DPL essentially provides deployment-enhanced architecture and composition information, the deployment catalog notation provides in addition to the DPL instances the constructs to enumerate the interfaces of a distribution pattern based deployment on a composition engine. It enables the composition engine to find the interfaces and dependent resources within a deployment archive. This notation abstracts specifications used by WS-BPEL engines such as ActiveBPEL to organise deployment information. It fills endpoint information with the corresponding address and service name data.

3.5 Discussion

Before describing a development scenario and the actual implementation, a key observation in relation to the transformation shall be discussed. In the transition from abstract to concrete, explicit information gets lost and is embedded into the created artifacts. Both distribution characteristics described through patterns, but also associated quality properties are affected:

– Distribution: Properties such as the node types that explain the role in a pattern become implicit, but visible (and not measurable) as structural topology properties of the executable system.

– Quality: Explicit attributes re-emerge as observable (sometimes measurable) quality properties (NFPs) of the executable system.

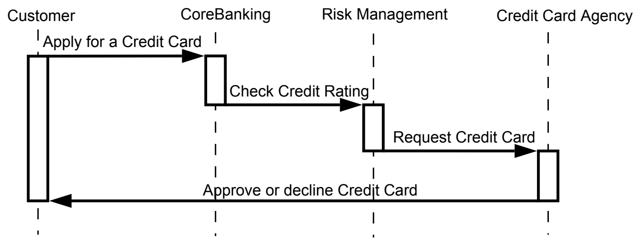

Figure 6: UML sequence diagram for bank case study

The core graph that defines composition and connectivity remains. It is important to preserve the structure in the transition from explicit to implicit as it defines the architecture.

4 Pattern-based Service Architecture Development

We illustrate the development of a service architecture based on the presented pattern techniques for a banking case study now, before widing the development perspective by discussing the link to quality-driven development in the presence of patterns. Quality considerations can determine the pattern to be selected, but can also help to refine existing patterns.

4.1 Case Study

An enterprise banking system with three interacting business processes shall illustrate the distribution requirements of a service architecture. 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. Figure 6 illustrates the interactions between a customer and the bank processes.

4.1.1 UML-based Distribution Modelling

The banking case study provides three WSDL interfaces as input to the model generator. These interfaces represent the bank (CoreBanking), the risk assessment agency (RiskManagement) and the credit card agency (CreditCard), see Figure 7. All three

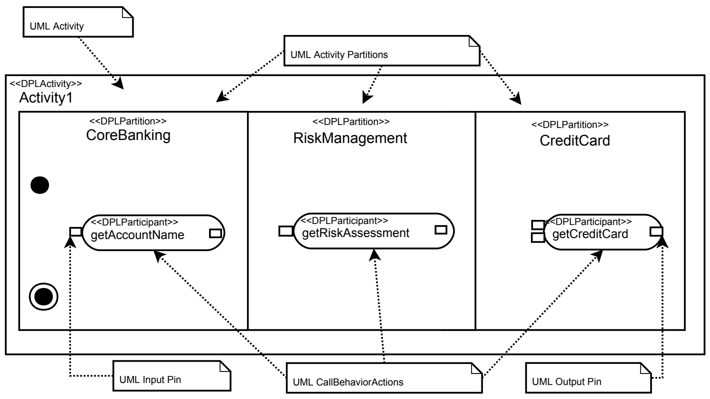

Figure 7: UML-based service integration for bank case study

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.

4.1.2 Distribution Pattern Selection

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 distribution pattern (with dedicated peers) 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. We do not use a dedicated peer as the entry point to the composition, although this option is available. To implement the pattern instance, 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, accountName, to the CreditCard CallBehaviorAction. Finally, appropriate values must then be assigned to the tagged values of the stereotypes. Some appropriate values can be seen in Table 1.

Table 1: DPLProfile stereotype attributes – graph attributes at meta-level (down to engine-uri) and service node types (from role onwards).

DPL Attribute	**Description**	**UML Base Element**	**Stereotype**
distribution-pattern	Choice of distribution pattern to be applied to composition	Activity	DPLMetadata
collaboration-language	Choice of collaboration language to enact composition	Activity	DPLMetadata
service-name	Name used by clients to reference the composition	Activity	DPLMetadata
base-namespace	Namespace URI for the composition, avoids name clashes	Activity	DPLMetadata
namespace-prefix	Namespace alias for the composition, avoids name clashes	Activity	DPLMetadata
operation-name	Operation name used by clients to reference the service	Activity	DPLMetadata
ns	Namespace URI of the participant, avoids name clashes	ActivityPartition	DPLPartition
interface-uri	URI specifying the location of the participant's interface	ActivityPartition	DPLPartition
engine-uri	URI specifying the location of the enactment engine	ActivityPartition	DPLPartition
role	Choice of roles for the participant from the Role enumeration	CallBehaviorAction	DPLParticipant
is-correlation-variable	Unique identifier field for a composition	Pin	DPLMessage
order	Execution order assigned to action	ControlFlow	DPLControlFlow

Finally, if the organisational structure of the bank is in flux, a peer-to-peer distribution pattern would provide a future-proof architecture in that peers can be swapped in and out based on new or changing business partners. A centralised distribution pattern would be appropriate if the bank controls all three services being composed and does not expect to receive a high number of credit card applications. This pattern is appropriate as the bank controls all the data being processed, i.e. there are no data security issues. The low number of expected credit card applications should not cause availability problems on the server.

4.1.3 Intermediate DPL Representation

Fig. 8 shows a DPL document instance for the case study. Three sections can be distinguished in the instance:

– It begins with the metadata attribute instantiation (`pattern_definition`) as defined in Table 1.

– Then, the nodes (`nodes`) declare the services involved.

– Finally, the mappings (`mappings`) connect the service to reflect interaction.

The message names and message parts have been truncated for space reasons. The document instance content is minimal to provide for easy parsing. A reference to the URI of the Web service interface is maintained in the document instance to allow for querying by other tools if necessary.

Many of the values in Fig. 8 are the same as the values applied by the software architect previously, 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.

4.1.4 Validation and Generation

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 – which are two examples of structural consistency problems. If any errors are detected they must be corrected by the software architect, usually by returning to the modelling stage.

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.

4.2 Distribution Patterns and Quality of Service

Different distribution patterns realise different non-functional properties. Quality of Service (QoS) is a term under which some of these properties are often grouped, see e.g. [19]. Categories of qualities which affect systems at runtime are for instance performance, dependability, and security [4]. These categories are of particular relevance for the distribution architecture of service-based systems. We have outlined some examples for our case study. We have taken some specific QoS characteristics applicable to distribution patterns from [27] and [30] to illustrate the importance of this connection.

– Performance - The timeliness with which a system can react to requests.

```
<?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 8: DPL document instance

- Latency - Speed of response to requests.
- Throughput - Number of requests handled within a period of time.
- Capacity - Number of requests that can be dealt with without a drop in throughput or increase in latency.

– Dependability - Measure of the reliance that can be placed upon a system functioning correctly.

- Availability - A measure of the system's up-time.
- Reliability - Ability of the system to remain available over time.
- Security - Protection of the system and its data from unauthorised use.
- Autonomy - Measure of independence of system's discrete parts.

The categories are used to associate qualities to distribution patterns and to classify or discover patterns based on these quality attributes [24]. However, it needs to be noted that this quality association is still an open problem in the research community.

To illustrate the principles, we look at the two most prevalent patterns – centralised and decentralised in more detail – before briefly describing the other patterns. The quality attributes of the core patterns which we discuss are documented by [12, 13, 44]. The centralised and decentralised patterns are the core patterns that embody two core principles of distribution:

– Centralised: In a centralised shared-hub pattern [12], 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 may suffer from a communication bottleneck at the central controller. This represents a considerable scalability and availability issue for larger enterprises.

– Decentralised: The decentralised shared-peer pattern [44] 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 [41]. 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.

Hub-and-Spoke and Peer-to-Peer are examples of centralised and decentralised patterns, respectively. These may be applied as variants of the first two distribution patterns when additional autonomy, scalability and availability is required. Other distribution patterns include 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 is applied. Complex patterns are useful in that the combination of patterns often results in the elimination of a weakness found in a core pattern. An example of a complex pattern is a "ring + centralised pattern", which provides clustering for a highly loaded central controller. For example, the addition of a dedicated hub to a centralised distribution pattern allows a composition to be initiated by a participant external to a composition. A similar scenario is where an additional peer is added to a decentralised distribution pattern to initiate a composition.

5 Implementation

TOPMAN (TOPology MANager) is our solution to enable the modelling of distribution topologies based on patterns using UML 2.0 and subsequent Web service composition generation based on the architecture outlined in Fig. 2 [5]. The tool utilises Eclipse technologies. As already emphasised, the integration and interoperability of the notations and techniques presented here with existing tool support for service architecture development is a crucial requirement for the applicability of our solution. Eclipse is the platform we are using to integrate modelling and transformation tools. Eclipse EMF editors are used to create ECore models:

- Modelling interface tools like RSA can be embedded to support modelling activities.
- ATL is the implementation language for the model transformations and the code generation.
- XML is the format to represent DPL instances.

Modelling and generation define two categories of tool components that shall now be discussed.

5.1 Modelling

The UML 2.0 model generator uses the transformation framework ATL to transform the WSDL interfaces of the Web services participants, represented in the form of class diagrams, to a UML 2.0 activity diagram [23]. The model generated includes a reference to our predefined UML profile for distribution patterns. We have encountered some issues during the development of our tool. As noted in [38], WSDL tools do not all generate WSDL interfaces according to the same naming conventions. Consequently, we must tailor the UML 2.0 model generator to specific WSDL tool idiosyncrasies.

A number of tools may be used to describe the distribution pattern. IBM's commercial tool Rational Software Architect (RSA) is compatible with UML 2.0 and supports many of the UML 2.0 features. The tool has a GUI which allows the software architect to define distribution architectures. Upon completion, the model can be exported back 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.

5.2 Generation

The distribution pattern generator uses ATL to transform the UML 2.0 model to a DPL instance document. The DPL document instance is then checked for correctness by an XML-based validating parser and semantics analyser. 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, ATL is 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 suitable 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 [13].

6 Evaluation

We have presented a pattern catalog, modelling and description notations, and conversion and generation techniques. To assess our approach, we use the criteria set out in [39], along with some of our own success criteria, for the respective components of our solution. A number of specific concerns need to be observed:

– Target user profile: Our approach, technique and implementation are targeted at software architects, who should be comfortable working with patterns and CASE-based tools which assist in the generation of executable systems.

– Tool integration: Our modelling effort is XMI-based and can therefore be integrated with both commercial and open source tools. Integration of the technique in existing development tool environments is paramount (thus our focus on notation and technique interoperability).

We apply different, quantitative and qualitative analysis techniques in our evaluation. The catalog is quantitatively evaluated with respect to usability. The language and techniques are looked at in terms of usability and correctness, which in particular in the case of correctness require qualitative methods.

6.1 Catalog

We have identified a number of distribution patterns, collected as a pattern catalog, and have shown how patterns can be expressed adequately using UML with our DPL-Profile extension and in XML, using DPL. The catalog is adequate in that 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. We have evaluated the catalog in terms of four measures: usage, coverage, utility and precision as outlined in [14]. We have evaluated the catalog based on five different application case studies [5]:

– Adding High Availability and Autonomic Behavior to Web Services
– Service Composition Modeling
– A p2p architecture for dynamic executing GIS web service composition
– Using a rigorous approach for engineering Web service compositions
– Migration to web services oriented architecture

The attributes usage, coverage, utility and precision result in the following measured values for the five applications:

– Usage: Ratio of pattern usage to total number of patterns in the catalog. A ratio of 7:4 (0.571) indicates that the majority of our patterns occur in common applications. The usage measure value of 0.571 shows that of the seven patterns in our catalog only four unique patterns were actually used in the case study applications. The centralised pattern occurs in two of in the case studies. No usage of three patterns, ring + centralised, centralised + decentralised and ring + decentralised was found in the case studies.
– Coverage: Ratio of pattern usage to total number of patterns in the application. A ratio of 5:5 (1.0) shows that patterns found in the applications were also part of our catalog.
– Utility: Average times a catalog is used in a given application. A ratio of 5:4 (0.8) indicates that some patterns such as the centralised ones are more prevalent than others. Of the four patterns used in the case study applications one is used twice, while three are used only once. This indicates, at least in our small use case sample, that the centralised distribution pattern is more prevalent than the other patterns. The highest value possible for this measure when considering architectural patterns is one.
– Precision: Ratio of pattern usage to number of adaptations required necessary to make the catalog useful. A ratio of 1:1 indicates that we found no reason to adapt patterns during our evaluation. However, it should be noted that architectural patterns do not normally require adaptation as they are at a very high level of abstraction.

In conclusion, we can say that the catalog is adequate as it provides common patterns, but also patterns for a range of rarer situations. The catalog provides a classified collection of reusable distribution topology solutions.

6.2 Languages and Techniques

Languages as well as validation and transformation techniques are the components of our solution. We look at selected properties of these in the context of our aim to provide a cost-effective interoperable service architecture method. An empirical evaluation using the TOPMAN tool and its integration into an existing tool landscape and the use of application case studies have complemented the analytical evaluation components.

- Language Usability: Our modelling approach, which visualises the distribution pattern, should be intelligible to software architects – which is achieved through a UML profile. As the model is platform-independent, implementation-level details are avoided. The independence of our approach from platform- and language-specific aspect makes it reusable. UML models and DPL instance documents are modelled at the platform-independent level, so there is no reliance on any collaboration language.
- Transformation and Validation Correctness: We have verified our model transformations using declarative QVT relations between corresponding meta-models. The graph-theoretic model additionally provides the link between the semantics of the individual notations and the transformation specification. In addition to preserving the functional semantics of the abstract architecture in the generated executable, our evaluations (in particular based on the case studies mentioned in the catalog evaluation and below in the context of maintainability) demonstrate that the qualities associated to the patterns are indeed transferred onto the executable system.

A central benefit of a more automated solution is a cost-effective development process. We have compared the architecture development with a manual architecture design and implementation. While coding is automated here, in the manual approach considerable service orchestration code (WS-BPEL and WSDL) has to be produced. In addition, maintenance is improved, further reducing costs. Our approach allows easy manipulation of the system's distribution pattern at a high level of abstraction. We have used the ALMA (Architecture Level Modifiability Analysis) method to evaluate maintainability [31]. Five change scenarios were selected: adding, removing and updating participants (3 scenarios), changing the distribution pattern, and changing the execution engine. Due to automation, all scenarios benefit from our approach. The most substantial benefit is gained in the context of changing a pattern. Here, the manipulation at the graphical, UML-based level simplifies the architect's work significantly. More details can be found in [5].

7 Related Work

We will now introduce some related work and contrast their approaches to our solution. Two workflow management systems shall be discussed that motivate and provide concrete implementations for two of the distribution patterns explored in this paper. However, neither system provides a standards-based modelling solution to drive the realisation of the chosen distribution pattern.

- The first system, DECS [44], from which the distribution pattern term originates, is a workflow management system, which supports both centralised and decentralised distribution patterns, albeit without a 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 [41], 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 [25] 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.

Grønmo et al. [39, 15] consider the modelling and building of compositions from existing Web services using MDA, an approach similar to ours. However, they consider only two modelling aspects, service (interface and operations) and workflow models (control and data flow concerns). The system's distribution pattern is not modelled, resulting in a fixed centralised distribution pattern for all compositions. Their modelling effort begins with the transformation of WSDL documents to UML, followed by the creation of a workflow engine-independent UML 1.4 activity diagram, 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 Modelling Notation (BPMN), instead of UML, for describing Web service processes [6]. 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 modelled.

A platform specific UML 1.4 based model is investigated in [21]. Here IBM's Rational Rose (now Rational Software Architect) is used to apply a UML profile to a WS-BPEL-based UML activity diagram. The model is capable of building a completely executable system based on MDA, albeit based only on a WS-BPEL workflow, as a platform specific model is used. As with Grønmo et al.'s work, distribution patterns are not considered by the authors. [2] propose a mapping from UML 2.0 sequence diagrams to a WS-BPEL workflow. The authors use MDA to drive the software development process and provide an example of a platform-independent model being transformed to a platform-specific model. While no executable system generation is proposed and distribution patterns are not considered, this approach is worth considering as its does

	Arief et al.	DECS	SELF-SERV	Web-ML	UMT	Topman
Web Service Support		Y	Y	Y	Y	Y
UML Model Support		Y	Y	Y	Y	Y
XML Model Support			Y	Y[2]	Y[2]	Y[2]
ADL Model Support			Y			
No. of Schemes Supported	n/a	2[1]	1	2	n/a	9
Models Architectures	Y					
Models Orchestrations		Y	Y	Y	Y	
Models Choreographies						Y
Code Generation Support	Y	Y	Y	Y	Y	Y
Dynamic Reconfiguration		Y				
Static Reconfiguration	Y		Y	Y	Y	Y

Table 2: Comparison of Related Approaches

(1) supports distribution scheme change, but not modelled explicitly; (2) XML support via XMI

not, in contrast with some other works, use UML activity diagram, but rather sequence diagrams.

Beyond modelling and generation, quality has been studied. [12] provides some interesting performance metrics to back up the use of decentralised execution instead of centralised execution. Here, we see fewer messages exchanged in a decentralised environment while execution time is also reduced for larger message sizes.

We have summarised the comparison with other frameworks in Table 2. We can see how our modeling and transformation framework TOPMAN compares with the feature set provided by the state of the art in modeling and transformation frameworks. Our approach, like a number of other tools, considers the Web service composition domain. We note that SELF-SERV is the only framework to support the use of an Architecture Description Language (ADL) as its modeling language. We chose not to use ADL as our modeling language because its various notations may not be as familiar to software architects as the UML 2 notation. We do, however, future proof our approach by defining our own DSL, the Distribution Pattern Language (DPL), which is not tied to UML 2, enabling future work to address a perceived need to model using ADL instead of UML 2. Some of the frameworks – DECS and SELF-SERV – provide only XML modeling of compositions. We believe that XML is not an ideal language for defining models because XML is overly verbose and is not human friendly when compared to a well designed visual representation of a composition. Our modeling approach using UML 2 provides a visual representation of the composition along with bindings to XML via the UML serialisation format XMI. This approach is also utilised successfully by UMT. An alternative approach to UML modeling is considered by WEB-ML where BPMN models are used instead of UML to model compositions. As previously noted, our approach could be amended to model with BPMN as we have ensured that our approach is not

UML dependent.

A number of the identified frameworks support different distribution patterns. However, only one of these approaches Web-ML, explicitly models the distribution pattern in a similar way to our approach. Web-ML is however restricted to only two distribution patterns. DECS also supports two distribution patterns; however it does not explicitly model the distribution pattern, and merely provide a switch to alter the behaviour of the executing system. Our approach, in contrast, considers seven distribution patterns.

The majority of the frameworks model compositions from an orchestration point of view. This perspective considers the workflow of the compositions participants rather than the non-functional requirements, which is our objective. It is worth noting that Arief et al. consider the modeling of system architectures using UML, an approach we also use. Also worthy of note is the Net Traveler approach, which considers both an orchestration and a choreography model, albeit using XML based models. Our modeling and transformation framework combines the UML modeling approach used by Arief et al. and the choreography models used by NetTraveler to address non-functional requirements of Web service compositions.

DECS is capable of altering the distribution pattern used at runtime. This functionality is desirable as it allows the distribution pattern to be changed in reaction to the execution environment. For example, under high load the decentralised distribution pattern is more favourable that the centralised pattern. We consider runtime alteration of distribution patterns on our modeling and transformation framework as future work.

8 Conclusion

An architecture approach to the composition of service-based software systems is required to overcome problems with ad-hoc and dynamic composition in recent service-based software systems. We have introduced techniques based on architectural modelling and pattern-based development, motivated by solutions successfully applied in both object-oriented and component-based systems. We have also applied patterns that have been found useful in a networking context to the Web service domain. Our contribution is a modelling and transformation technique for expressing the distribution pattern-based architecture of service compositions. Our novel modelling 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. Our approach is characterised by a stepwise transformation approach, which makes initially explicit distribution topology characteristics and distribution architecture qualities implicit, but observable aspects of the generated system.

A number of modelling and transformation techniques were introduced, along with a tool (TOPMAN) which assists in the generation of an executable system guided by the chosen pattern. A UML 2.0 activity diagram and profile extension to model the distribution pattern, a modelling technique independent of the document format for specifying and validating distribution patterns and finally a number of generators for transforming models from one format to another and subsequently generating an executable system, considerably reduce the coding effort and workload of a software architect.

Interoperability with existing tools and integration of different notation has been a primary goal. Integration is central for two reasons:

– We can expect existing service architecture support in various ways. For instance service implementations such as Java EJBs could be exposed Web services in an eclipse-based development environment. Other tools might support domain and data modelling. Our tools would need to be integrated with these.

– We intend considering integrating alternatives to our UML modelling language approach, based on the π-calculus and Architecture Description Languages. A model generator for these modelling languages would extend the reach of our approach.

Interoperability and integration could be further enhanced by supplementing the presented architecture approach by semantic modelling and reasoning techniques. In [35, 36], we have investigated the possibility of using ontologically enhanced semantic descriptions of services to support service discovery and matching in the context of process composition. Ontology-based reasoning increases the degree of automation [33]. While the service composition benefits in principle as we demonstrated, the presented techniques would need to be adapted to the pattern descriptions here, i.e. the UML profile and the DPL would need to be extended.

We have added a discussion of quality and architectures in the development section, which aims to emphasise the increasing importance. The documentation of the QoS attributes of the complex patterns is an important future effort, which can act as a starting point for further investigations that are beyond the scope of this work. Links between architectural patterns and quality attributes have been discussed in the literature. A conclusive evaluation of the success of quality-driven architecture has to be looked at in more detail – providing in particular empirical evidence of the establishment of quality attributes through patterns.

Acknowledgment

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

References

1. Alonso, G., Casati, F., Kuno, H., and Machiraju, V.: *Web Services: Concepts, Architecture and Applications* Springer Verlag, 2004.
2. B. Bauer and J. Müller: MDA Applied: From Sequence Diagrams to Web Service Choreography; In *Proc. 4th International Conference (ICWE 2004)*, pages 132–136, Munich, Germany, July 2004.

3. Bézivin, J.: In Search of a Basic Principle for Model Driven Engineering; *UPGRADE*, 2, 2004.
4. Barbacci, M.: Quality Attributes; Technical report, CMU/SEI-95-TR-021, 1995.
5. Barrett, R.: *Investigations into the Model Driven Design of Distribution Patterns for Web Service Compositions* Ph.d. (thesis), Dublin City University, Dublin 9, Ireland, 2008.
6. Brambilla, M., Ceri, S., Fraternali, P., and Manolescu, I.: Process Modeling in Web Applications; *ACM Transactions on Software Engineering and Methodology*, 15(4):360–40, 2006.
7. Baldan, P., Corradini, A., and Gadducci, F.: Specifying and verifying uml activity diagrams via graph transformation; In *Global Computing, LNCS 3267*, pages 18–33, 2005.
8. Buschmann, F., Henney, K., and Schmidt, D.: *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing* Wiley, 2007.
9. Barrett, R. and Pahl, C.: Model Driven Design of Distribution Patterns for Web Service Compositions; In *The International Workshop on Models for Enterprise Computing (IWMEC 06)*, Hong Kong, China, October 2006.
10. Barrett, R. and Pahl, C.: Semi-Automatic Distribution Pattern Modeling of Web Service Compositions using Semantics; In *Proc. Tenth IEEE International EDOC Conference*, Hong Kong, China, October 2006.
11. Barrett, R., Pahl, C., Patcas, L., and Murphy, J.: Model Driven Distribution Pattern Design for Dynamic Web Service Compositions; In *Proc. Sixth International Conference on Web Engineering*, Palo Alto, USA, July 2006.
12. Benatallah, B., Sheng, Q., and Dumas, M.: The Self-Serv Environment for Web Services Composition; *IEEE Internet Computing*, 7:40–48, 2003.
13. Chafle, G. B., Chandra, S., Mann, V., and Nanda, M. G.: Decentralized orchestration of composite web services; In *Proc. 13th international World Wide Web conference*, pages 134 – 143, New York, NY, USA, May 2004.
14. Cutumisu, M., Onuczko, C., Szafron, D., Schaeffer, J., McNaughton, M., Roy, T., Siegel, J., and Carbonaro, M.: Evaluating Pattern Catalogs: The Computer Games Experience; In *Proc. 28th international conference on Software engineering (ICSE'06)*, pages 132–141. ACM Press, 2006.
15. 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.
16. Ding, C., Nutanong, S., and Buyya, R.: P2P Networks for Content Sharing; *CoRR*, cs.DC/0402018, 2004.
17. Engels, G., Hausmann, J., Heckel, R., and Sauer, S.: Testing the consistency of dynamic uml diagrams; In *Proc. Int'l Conf. IDPT*, 2002.
18. Eriksson, H., Penker, M., Lyons, B., and Fado, D.: *UML 2 Toolkit* Wiley, 2003.
19. Frolund, S. and Koistinen, J.: 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.
20. Frankel, D.: *Model Driven Architecture: Applying MDA to Enterprise Computing* Wiley, 2004.
21. Gardner, T.: 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.
22. Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: *Design Patterns:Elements of Reusable Object-Oriented Software* Addison-Wesley, 1995.
23. Grose, T.: *Mastering XMI: Java Programming with XMI, XML, and UML* Wiley, 2002.
24. Harrison, N. and Avgeriou, P.: Leveraging Architecture Patterns to Satisfy Quality Attributes; In *Proc. 1st European Conference on Software Architecture*, Madrid, Spain, 2007. Springer Verlag.
25. 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.

26. Koch, N.: Transformations Techniques in the Model-Driven Development Process of UWE; In *Proc. of 2nd Model-Driven Web Engineering Workshop*, Palo Alto, USA, July 2006.
27. Menascé, D.: QoS Issues in Web Services; *IEEE Internet Computing*, 6(6):72–75, November–December 2002.
28. OMG: Unified Modeling Language (UML), version 2.0; Technical report, OMG, 2003.
29. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Final Adopted Specification; Technical report, OMG, 2005.
30. OMG: UML Profile for QoS and Fault Tolerance; Technical report, OMG, 2005.
31. P. Bengtsson and N. Lassing and J. Bosch and H. van Vliet: Architecture-Level Modifiability Analysis (ALMA); *Journal of Systems and Software*, 69(1-2):129–147, 2004.
32. Pahl, C.: A formal composition and interaction model for a web component platform; *Electronic Notes in Theoretical Computer Science*, 66(4):67 – 81, 2002 Formal Methods and Component Interaction (ICALP 2002 Satellite Workshop).
33. Pahl, C.: A conceptual architecture for semantic web services development and deployment; *Int. Journal of Web and Grid Services*, 1(3/4):287–304, 2005.
34. Pahl, C.: Layered ontological modelling for web service-oriented model-driven architecture; In *European Conference on Model-Driven Architecture ECMDA2005*, pages 88–102. Springer LNCS Series, 2005.
35. Pahl, C.: Semantic model-driven architecting of service-based software systems; *Information and Software Technology*, 49(8):838–850, August 2007.
36. Pahl, C. and Barrett, R.: An Ontological Framework for Web Service Processes; *International Journal of Software Engineering and Knowledge Engineering*, 18(3):383 – 411, 2008.
37. Peltz, C.: Web Services Orchestration and Choreography; *IEEE Computer*, 36, 2003.
38. R. Grønmo and D. Skogan and I. Solheim and J. Oldevik: Model-Driven Web Service Development; *The International Journal of Web Services Research*, 1, 2004.
39. 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.
40. Rensink, A. and Nederpel, R.: Graph transformation semantics for a qvt language; *Electron. Notes Theor. Comput. Sci.*, 211:51–62, 2008.
41. Sheng, Q., Benatallah, B., and Dumas, M.: 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.
42. Tsiolakis, A. and Ehrig, H.: Consistency analysis of uml class and sequence diagrams using attributed graph grammars; In *Proc. of Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems*, pages 77–86, 2000.
43. Taylor, R., Medvidovic, N., and Dashofy, E.: *Software Architecture: Foundations, Theory, and Practice* Wiley, 2009.
44. Woodman, S. J., Palmer, D. J., Shrivastava, S. K., and Wheater, S. M.: A System for Distributed Enactment of Composite Web Services; In *Work in progress report, Int. Conf. on Service Oriented Computing*, Trento, Italy, December 2003.