Quality-aware Model-driven Service Engineering[1]

Claus Pahl[1], Marko Bošković[2], Ronan Barrett[1], Wilhelm Hasselbring[2]

[1] Dublin City University
School of Computing
Dublin 9
Ireland

[2] University of Oldenburg
Department Informatik
Oldenburg
Germany

Abstract. Service engineering and service-oriented architecture as an integration and platform technology is a recent approach to software systems integration. Quality aspects ranging from interoperability to maintainability to performance are of central importance for the integration of heterogeneous, distributed service-based systems. Architecture models can substantially influence quality attributes of the implemented software systems. Besides the benefits of explicit architectures on maintainability and reuse, architectural constraints such as styles, reference architectures and architectural patterns can influence observable software properties such as performance. Empirical performance evaluation is a process of measuring and evaluating the performance of implemented software. We present an approach for addressing the quality of services and service-based systems at the model-level in the context of model-driven service engineering. The focus on architecture-level models is a consequence of the black-box character of services.

Keywords. Service-oriented architecture, model-driven development, software product quality, architectural modelling, patterns, empirical performance evaluation.

1) Introduction

With software services becoming a strategic capability for the software sector, a service engineering discipline needs to address service development problems based on suitably flexible modelling and composition support. An increasing need for flexibility in this area is caused by changing user requirements, evolving services, and varying deployment

[1] ***This chapter appears in "Model-Driven Software Development: Integrating Quality Assurance" edited by Rech, J and Bunse, C, Copyright 2008, IGI Global, www.igi-global.com. Posted by permission of the publisher.***

contexts. Software services are application that are provided ‘as-is’ at certain locations in order to be integrated into existing applications or composed to larger systems. Essential in this process are abstract descriptions or models of the service functionality and other service characteristics. This makes model-driven software development both a highly suitable, but actually also necessary framework to adequately develop service-based software systems. Composition and integration-oriented modelling has already been successfully utilised for model-driven services development. The high complexity of modern software makes its development costly and error-prone. Model-driven development (MDD) is an approach that deals with software complexity by making software models primary artefacts of the software development process. MDD utilises two aspects of models. Firstly, in various engineering disciplines, predictions about a software system can be made based on a model. Secondly, even complete implementations for different platforms and languages can be generated from models.

The aim of this chapter is to address quality aspects of model-driven service engineering. We address two specific facets of quality assurance for model-driven software development: the model-driven design and development of high-quality software and the identification of quality aspects in model-driven development. Some specific aspects that we discuss in the context of quality aspects in model-driven service development are:

- modelling and architecture are concept that are closely linked in the context of model-driven development and service-oriented architecture,
- patterns for the model-driven development of service architectures to structure models and to enhance the integration task,
- model-based design and analysis of specific quality aspects for service-based systems.

We aim to demonstrate that model-driven architecture and design of this specific type of service-based software systems can yield high-quality software. Based on an analysis of the state of the art of service engineering and its platform and application requirements supported by a case study analysis, we identify the central quality aspect pertinent to services. This analysis is necessary in order to justify techniques for the quality-aware model-driven service engineering. Factors that impact the quality are:

- network and platform characteristic impacting on for instance performance,
- service-orientation to enhance interoperability and reusability,
- evolution and change as inevitable factors impacting on maintainability.

Service engineering and service-oriented architecture as an integration and platform technology is a recent approach to service-based software systems integration. Quality aspects, however, have not been addressed in sufficient depth in this context. Our technical contribution is an architecture- and pattern-based model-driven service engineering framework that aims at high-quality models as well as quality implementations. While functionally oriented design patterns have been widely used to support the development of large-scale software systems, we combine a service-specific range of these patterns with distribution patterns, which directly impact service-specific quality aspects such as performance. We complement this architectural perspective with an empirical, model-driven performance evaluation technique. To provide trustworthy software, quality attributes have to be satisfied. Performance is a quality attribute that shows the degree to which a software system or its components meet the objectives for timeliness. Quality attributes such as performance are of central importance for the integration of heterogeneous, distributed service-based systems.

We start this investigation with an introduction to model-driven development and service engineering in Section 2. Quality aspects are outlined in Section 3. Architecture modelling notations are introduced in Section 4 and applied in the service architecture context in Section 5. Distribution patterns as an MDD solution are discussed in Section 6 and model-driven performance evaluation is investigated in Section 7.

2) Model-driven Service Engineering

Service-oriented architecture as the architectural methodology and the Web Services as the deployment platform have implications on a corresponding service engineering framework. This section motivates model-driven service engineering as our framework:

- We introduce service engineering as a software engineering discipline and service-oriented architecture as an architectural framework.
- Web services as a specific platform for service-oriented architecture within a service engineering context are introduced.
- An overview of model-driven service development aspects concludes this section.

This section aims to clarify the principles of service-based software systems and their development support through model-driven approaches.

2.1) Service Engineering

A service in the context of service-oriented architecture (SOA) is a software component provided at a given location. Services are usually used 'as-is', based on an abstract description published by the service provider in directories and used by potential clients to locate suitable services. Several aspects characterise the targeted service platform (Alonso et al., 2004):

- Distribution: This deployment aspect characterises the services platform as a distributed infrastructure.
- Independent deployment: This development aspect refers to the independent, black-box deployment of services where different organisations are involved as clients and providers. This requires suitable description techniques to communicate service requirements and properties.
- Process-orientation: The development of services is tightly linked to the notions of architecture and process-centric configuration. The composition at various levels

ranging from business workflows to service processes is central (Allen & Garlan, 1997; Plasil & Visnovsky, 2002). Orchestration and choreography are two perspectives on service process composition (Alonso et al., 2004).
- Software value chain: The notion of a software value chain emphasises the step-wise process of software development based on layered modelling techniques covering all stages from client-orientation to implementation by adding to models until a deployable service product is realised (Weber, 2005).

The composition of services to orchestrated processes is a major concern in current Web service research (Allen & Garlan, 1997; Plasil & Visnovsky, 2002; Bass et al., 2003). These developments have strengthened the importance of architectural questions such as service composition and configuration.

Model-driven development (MDD) emphases automation and encourages model reuse. Service-oriented architecture focuses on reuse-as-is in service form. We propose here central cornerstones of a model-driven development framework for service-based software. Composition-centric modelling shall address services, processes, and layered reuse in order to improve quality for instance in terms of maintainability. We have identified a number of aspects by reviewing case studies, which are facets that characterise the framework and that reflect factors that have impacted the design of our proposed development approach.
- Rigorous and formal foundations – the foundations of the modelling notation and techniques in terms of formal models.
- Service development and deployment process – the software process lifecycle with its stages and activities based on MDD.
- Development methods and techniques – composition-centric modelling and service architecture to support the activities.
- Standards and interoperability – the technology environment with its opportunities and constraints focusing on model and service interoperability.

2.2) Model-Driven Engineering

The general idea of model-driven development (MDD) is to introduce a model as a first-class entity. With models, the development focus is moved to the problem domain. Models often enable the exploitation of formal mathematical methods. With abstraction, the understanding of the problem and its realisation in a software system can be improved. Often, a complete implementation can be generated without discontinuities (Selic, 2003). Model Driven Architecture (MDA) is one approach for MDD initiated by the OMG (OMG, 2003), a consortium of software vendors and users. The MDA initiative consists of three complementary ideas (Selic, 2003):
- direct representation to shift the focus of software development away from technology toward the ideas and concepts of the problem domain,
- automation to mechanize the relation of semantic concepts from a problem domain and from an implementation domain,

- open standards to enable interoperability, often in an application-specific context to close the semantic gap between domain problems and implementation technologies.

Our aim is to enable the evaluation of service performance when the primary artefact is a service (or service process) model.

2.3) Model-Driven Service Engineering

Modelling can support architectural questions that arise in service-oriented architecture. Behaviour and interaction processes are central modelling concerns for service-based software architectures. Fig. 1 illustrates how a UML activity diagram can be used to express a service orchestration. Four services that provide e-learning activities – system login, lecture participation, lab participation, and system logout – are orchestrated into a process starting with a login, then allowing a user to iteratively choose between lecture and lab activities, before logging out.

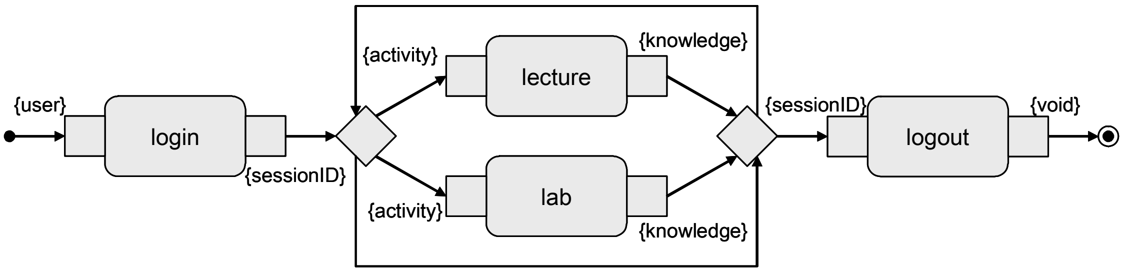


Figure 1. Service Process Modelled using a UML Activity Diagram.

Explicit descriptions and exchangeable models enable developers and clients of services to create reliable service architectures using tool support. A model-driven development approach can even be utilised to support automated code generation and performance analysis. Assuming that concrete, provided services are already attached to each service element, then an executable WS-BPEL process for the Web service platform can be automatically generated. As we are going to demonstrate later on, the service composition model can be instrumented for empirical performance analysis and executable processes including performance monitoring functionality can be generated.

2.4) Model-driven Service Engineering Case Study

The techniques of our framework shall be illustrated in the context of a re-engineering of the IDLE case study system as a service-based software system. IDLE is the Interactive Database Learning Environment – a Web-based learning and training system for database modelling and programming (Pahl et al., 2004). IDLE is a challenging application in terms of its need to provide multimedia and interactivity across a distribute server and user base. It allows us to focus on three essential quality aspects of service-based software systems: performance, reusability, and maintainability. We illustrate quality-aware model-driven service engineering using a recent re-engineering of IDLE as a fully service-oriented infrastructure.

3) Quality in Service-based Software Systems

An analysis of the IDLE case study software system and other systems based on a literature review shall clarify quality considerations in the context of model-driven service engineering. This section

- aims to identify system-critical quality aspects and their determining factors from an empirical analysis of IDLE,
- evaluates the IDLE case study findings in the context of existing literature and discusses the requirements for quality-aware model-driven service engineering,
- introduces quality aspects in software systems, focusing on maintenance and performance as two specific quality attributes.

As individual quality attributes vary in the way they can be described, measured and analysed, we are not able to discuss all attribute-specific techniques in detail. We have selected attributes relevant to SOA as an integration technique – such as maintainability, reusability, interoperability and performance – and illustrate the common and generic features of our framework using these attributes. Before looking at IDLE, we introduce the quality and performance principles.

3.1) ISO 9126 Software Engineering – Software Product Quality

The ISO/IEC 9126 standard on software product quality (ISO, 2001) defines a two-part model for software product quality consisting of quality attributes (internal quality and external quality) and quality in use aspects.

- The first part of the model specifies six characteristics for internal and external quality, which are further subdivided into subcharacteristics. The six categories are functionality, reliability, usability, efficiency, maintainability, and portability. The subcharacteristics are manifested externally when the software is used as a part of a computer system, and are a result of internal software attributes.
- The second part of the model specifies four quality in use characteristics. The quality in use characteristics are effectiveness, productivity, safety and satisfaction. Quality in use is the combined effect for the user of the six software product quality characteristics.

In principle, the internal quality determines the external quality and external quality determines quality in use.

- Internal metrics measure the software itself. Internal metrics are those which do not rely on software execution (static measures).
- External metrics measure the behaviour of the computer-based system that includes the software. External metrics are applicable to running software.

- Quality in use metrics measure the effects of using the software in a specific context of use. Quality in use metrics are only available when the final product is used in real conditions.

3.2) IDLE Software Quality

We use the IDLE system to illustrate our approach (Pahl et al., 2004). IDLE is based on a Web software architecture that provides a range of educational services:

- It is a multimedia system that uses different mechanisms to provide access to learning content, e.g. Web server and a (synchronised) audio server.
- It is a composite, interactive system that integrates components of a database development environment (a design editor, a programming interface, and an analysis tool) into a teaching and learning context.
- It is a constructive environment in which learners can develop their database applications, supported by shared storage and workspace facilities and knowledge-level interactions between learner and system.

A comprehensive discussion of all quality aspects is beyond the scope of this chapter. However, some specific aspects shall be singled out.

- Portability. In particular interoperability shall be addressed. We identify two dimensions – generic, often middleware-induced and domain-specific – that are relevant for IDLE. On the architectural levels these are captured through architectural styles and reference architectures. Architectures suitable to be supported by the Web services platform need to be considered. Additionally, in order to allow specific components to be reused or exchanged, compliance with domain-specific reference architectures such as the Learning Technology System Architecture standards (LTSA) might be required. A current problem is the integration of components and content of the system into a national learning resource repository that allows these resources to be shared among third-level institutions. Interoperability is an absolute necessity, but other qualities such as reliability and performance are important as well.
- Maintainability. IDLE is a system that has been developed over a period of more than 10 years in several phases. Maintainability has been a critical problem due to high fluctuation of developers and an often experimental style of development driven by research aspects. Model-driven development can here be a contributor to improved maintainability. A number of different Internet-based technologies have been used in IDLE's implementation, ranging from audio processing with specific formats and servers to Java servlet-based middleware applications and storage solutions. An integrated infrastructure based on Web services is a current re-engineering problem in order to achieve maintainability and scalability. Knowledge-level interactions of users with the system using subject-specific editors and processing tools have equally made maintenance difficult. Only semantically enhanced information architectures and models can provide solutions here.

- Efficiency. IDLE as a bandwidth-demanding, distributed service-based Web software system is due to its needs arising from distributed multimedia delivery of content a system where performance is a critical factor. Performance evaluation – an aspect of the efficiency category – is important in order to provide users with a satisfactory usage experience.

In other distributed and Web-based applications, other qualities might be equally important – security and availability are two typical examples – but we have decided to focus in the ones relevant to IDLE, since a comprehensive account is beyond the scope of this chapter.

3.3) Discussion of Quality Aspects

Quality in the context of service-based software development is different from traditional software development and implementation. Services are black-box entities, i.e. they are used as-is. Determining or observing qualities through the inspection of service internals is in general not possible and would violate the principle of service computing. Consequently, quality needs to be considered at the architectural level. In particular, service compositions and service activations across a network, possibly distributed infrastructure are important and determine the categories we have singled out, i.e. portability, maintainability, and efficiency.

We can identify a number of central quality aspects pertinent to services. Factors that impact the quality are:

- internal, i.e. development and lifecycle-oriented static attributes:
    - portability and reusability as a consequence of service-orientation and architectural style compliancy,
    - maintainability as a consequence of model-driven controlled evolution and change,
- external, i.e. deployment- and execution-oriented observable attributes:
    - performance as a consequence of for instance network characteristics,

Specific techniques of model-driven service engineering in the context of those specific quality categories are:

- modelling of architecture constraints in the form of styles and reference architectures addressing portability,
- patterns for structured model-driven development addressing maintainability,
- analysis and evaluation of quality-aware models addressing performance.

A sufficiently rich and tailored modelling language is necessary to adequately address and achieve quality for service-based systems at the architecture model level, which is outlined in the next section.

4) Architectural Modelling

Before addressing the specific aspects architecture constraints, architecture patterns and performance evaluation, we introduce some basic elements of a notation that we use for architecture modelling.

The objective of software architecture (Bass et al., 2003) is the separation of computation and communication. Architectures are about components (i.e. loci of computation) and connectors (i.e. loci of communication). This allows a developer to focus on structures and the dynamics between components separately from component implementation. Various architecture description languages (ADLs) and modelling and development techniques have been proposed (Medvidovic & Taylor, 1997; Garlan & Schmerl, 2006; Cuesta et al., 2005; Oquendo et al., 2005). An architectural model captures common concepts found in a variety of architectural description languages: components provide computation, interfaces provide access to black-box components, and connectors provide connections between components.

Although UML is the most widely used modelling language, we use a mix of notations here. Textual notations are common for architectural description languages; we use them for instance for specific aspects such as architectural types and interaction behaviour. UML is the predominant notation for model-driven development; we use UML often as a visualisation of textual model specifications to emphasise the link to model-driven development. We use the textual notations as they demonstrate the link to formal specification notations and their underlying mathematical theories, such as process calculi or description logics, which is important if reasoning capabilities are to be exploited.

4.1) An Architecture Ontology

At the core of the notation is an architecture ontology that defines the central concept of the modelling notation. The central concepts are five core types of architectural elements

Configuration, *Component*, *Connector*, *Role*, *Port*

These are all derived from a general concept called *Element* that captures all architectural notions.

Components und connectors are the core elements of architecture descriptions. Components encapsulate computation and connectors represent communication between the components. Components can communicate through ports. Connectors connect to these ports, whereby the connection can play a specific role. Configurations are compositions of components and connectors with their ports and roles.

The vocabulary of the five elements shall be defined formally in terms of a simple logical formulation. This is loosely based on description logics, which often act as formal models of ontology languages such as the Web Ontology Language OWL.

$$Component \vee Connector \vee Role \vee Port \vee Configuration \subseteq Element$$

and

$$\begin{aligned}
Configuration &= \exists\ hasPart\ .\ (Component \vee Connector \vee Role \vee Port) \\
Component &= Element \wedge \exists\ hasInterface\ .\ Port \\
Connector &= Element \wedge \exists\ hasInterface\ .\ Role
\end{aligned}$$

The subset relation expresses subsumption, i.e. the subclass-superclass relationship. The predicates hasPart and hasInterface are predefined relationships between architecture elements. For instance, a configuration has parts such as component, connector, role or port. The existential qualifier describes that these components might exist. In terms of architecture models, these elements are types, i.e. are meta-level constraints for a concrete architecture.

4.2) Service Architectures, Processes and Dynamic Dependencies

Process and interaction behaviour is an essential part of modelling software architectures (Plasil and Visnovsky, 2002), in particular for service-based software systems. Interaction processes are central for the understanding of the behaviour of a software system. For instance, (Kazman et al., 2000) use scenarios – descriptions of interactions of a user with a system – to operationalise requirements and map these to a system architecture. We have extended the notion of interaction and also considered system-internal interactions. We also allowed interaction processes to be composite. Interaction process descriptions are forms of dependencies between components based on the connectors that need to be captured and addressed explicitly.

A service is defined as a coherent set of operations. An abstract service interface description is usually available. More recently, research has focussed on the composition of service to processes (Alonso et al., 2004). Existing components can be reused and assembled to form business or workflow processes. The principle of architectural composition that we look at here is process assembly.

4.3) Architecture Modelling Notation

We introduce a notation for the architectural modelling of service compositions and interaction that extends the previous structural focus of the architecture ontology. Our objective is to identify features of an architectural engineering language for services. This could be mapped or embedded into a full-scale architecture description language (ADL). Interaction behaviour for architecture configurations is an important feature for service architectures. Process calculi are often used in ADLs to express this type of information. Two elements define our notation.

- Firstly, a description notation is needed to capture architectural properties of a service-based software system.

- Secondly, the notation is complemented by modelling and analysis techniques.

The notation is defined in terms of the π-calculus (Sangiorgi & Walker, 2001) for two reasons. Firstly, a simulation notion allows us to formalise the transformation into executable code – for instance for empirical performance evaluation and instrumentation. Secondly, mobility is similar to changes in context – which gives us a formal framework in which to define change and address maintainability. We have introduced a new notation in order to tailor the π-calculus: firstly, to hide some of the more mathematical constructs and, secondly, to provide a focused set of operators for service composition.

The basic element describing process activity is an action. Actions are combined to process expressions. Given a service *x* and data item *a*, actions can be divided into invocations ***inv*** *x (a)* of other services and activations receive ***rcv*** *x (a)* and reply ***rep*** *x (b)* through other services. The process combinators are:

- Sequences are represented by the ';'-notation *a;P*, meaning that action *a* is executed and the system transfers to the remainder in the form of process *P* where the next action of *P* is executed.
- Choice means that one a_i from ***choice*** $a_i;P_i$ ($i=1,..,n$) is chosen.
- Multichoice ***mchoice*** $a_i;P_i$ (i=1,..,n) allows any number of the processes $a_i;P_i$ to be chosen and executed in concurrently.
- Iteration ***repeat*** *P* executes process *P* an arbitrary number of times.
- Parallel composition ***par*** (P_1,P_2) executes processes P_1 and P_2 concurrently.

Additionally, process abstractions shall be introduced. The following example introduces *Coach* as an abstraction, which is defined as a repeated choice of three individual actions (one of which is a sequence). Results of invocations can be assigned to variables.

Coach := ***repeat*** (***choice*** (**rcv** *getPref*(); **rep** *getPref*(*prefInfo*),
rcv *setPref*(*prefInfo*), *uri* = **inv** *locator* (*resource*)))

We cover the different aspects of architectural interaction modelling with this notation. Workflow operators are directly integrated as operators. An architectural design pattern can be formulated as an expression of a number of concurrently executing processes. Reference architectures and styles can be modelled on the level of abstractions in terms of the architecture ontology.

The architecture modelling notation is complemented by modelling and analysis techniques that suit the architectural engineering needs. A notion of satisfaction is needed to capture equivalence and refinement – an essential element of the modelling aspect. A simulation definition, adopted from the π-calculus, satisfies this requirement for the processes. A simulation needs to match all actions of the original process in the same ordering.

4.4) Modelling Approach – Overview

We have used textual notations for both the architecture ontology and also the behavioural specification constructs. Both could have been represented in terms of UML diagrams such as class and activity diagrams, with some OCL extensions. In order to clarify the formal background of these aspects, we have opted to represent them in terms of logics and process calculi notations.

The outline modelling notation support the three aspects identified earlier on. These three will be addressed in depth in the next three sections.

- A Styles and Reference Architectures Section aims at the internal attributes portability, interoperability and also reusability questions at the design level. Domain-specific and middleware-induced architectural constraints will be investigated.
- A Patterns Section looks at patterns in the form of architectural, workflow and distribution patterns with two aims: firstly, to deal with maintainability at design level; secondly, to address performance and in principle also reliability, availability and other external attributes at the code level.
- An Instrumentation Section looks at performance as a specific external attribute at the code level. Explicit instrumentation is an addition to the architecture modelling in terms of constraints and patterns.

Deployment and execution of instrumented code will also be addressed briefly, although our focus is on the modelling aspects.

5) Service Architecture – Styles and Reference Architectures

A central contributor to quality at architectural level is reuse. The reuse of architecture and components leads to better architectures in two aspects:

- maintainable through well understood structural and behavioural aspects,
- interoperable through standards-compliancy in terms of architectural aspects.

Architectural styles and reference architectures are related concepts that both constrain architectures, although with different objectives.

- Architectural Styles. We look at generic styles, like pipe-and-filter, which introduce a vocabulary of architectural element types with specific structural properties in terms of connectivity. The aim is to support interoperability and reuse from an internal, middleware- and platform-oriented perspective.
- Reference Architectures. We look at domain-specific reference architectures such as the IEEE Learning Technology System Architecture LTSA for learning technology systems. Their aim is often to support interoperability and reuse from an external, cross-organisational perspective.

The Architecture Ontology from Section 4.1 provides here the central notation.

Software architectures often act as a bridge between the client-oriented requirements and the software implementation-oriented design stages. The architectural style language for service architectures is directly based on the architecture ontology presented earlier. The style language is structural and connectivity-oriented. Styles are abstract models that aim to either

- reflect quality high-level design of software systems, i.e. aim typically at internal quality attributes, or
- constrain towards specific middleware and platform technologies, i.e. aim at interoperability.

They are therefore an integral part of a quality-aware model-driven service architecture framework.

5.1) Architectural Styles Principles

Architectural styles are reusable, recurring patterns in software architectures that are proven to have specific quality attributes (Abowd et al., 1995; Spitznagel et al., 1998; Baresi et al; 2004; Cortellessa et al., 2006; Giesecke, 2006). Typical examples are client-server, n-tiered, or pipe-and-filter architectures. These architectures share a common vocabulary, defining the elements of the architecture, and common constraints, defining the structural and behavioural restrictions that might apply.

Most ADLs focus on the component and connector view, i.e. model a system in terms of the components that will be implemented and executed and their connectivity. An architectural style language for this context needs to provide a type language – such as our architecture ontology – that allows basic element types – such as component, connector, or port – to be instantiated. The possibility to augment these types by structural and behavioural constraints is another necessary part of a style language.

5.2) Architectural Style Modelling

Defining architecture styles is actually done by extending the basic vocabulary of core types from the architecture ontology. The subsumption relationship serves to introduce the specific types that form an architectural style. This shall be illustrated using the pipe-and-filter style. We start with an extension of the hierarchy of architecture types in order to introduce style-specific components and ports:

$$\begin{array}{lll} \mathit{PipeFilterComponent} & \subseteq & \mathit{Component} \\ \mathit{PipeFilterPort} & \subseteq & \mathit{Port} \end{array}$$

These new elements shall be further detailed and restricted to express their semantics. We distinguish three types of pipe-filter components, *DataSource*, *DataSink* and Filter. Their respective connectivity through input and output ports is defined as follows:

$$\mathit{DataSource} \quad = \quad \leq 1\ \mathit{hasPort} \wedge \exists\ \mathit{hasPort}\ .\ \mathit{Output}$$

$$\begin{array}{lll} \mathit{DataSink} & = & \leq 1\ \mathit{hasPort} \wedge \exists\ \mathit{hasPort}\ .\ \mathit{Input} \\ \mathit{Filter} & = & = 2\ \mathit{hasPort} \wedge \exists\ \mathit{hasPort}\ .\ \mathit{Input} \wedge \exists\ \mathit{hasPort}\ .\ \mathit{Output} \end{array}$$

DataSource, *DataSink*, and *Filter* are defined as components of a pipe-and-filter architectural style. Each of these components is characterised through the number and types of component ports using so-called predicate restrictions on a numerical domain and the usual concept descriptions. The expression $\leq n$ is used to express $\mathit{hasPort}.(\ n \mid n \leq 1)$ for instance. In addition to these more structural conditions that define the connections between the component types, a number of semantic constraints can be formulated that further refine the initial enumeration of pipe-filter components.

- Disjointness requires the individual components to be truly different:

$$\mathit{DataSouce} \wedge \mathit{DataSink} \wedge \mathit{Filter} \quad = \quad \perp$$

- Completeness requires pipe-and-filter components to be made up of only the three specified types:

$$\mathit{PipeFilterComponent} \quad = \quad \mathit{DataSource} \vee \mathit{DataSink} \vee \mathit{Filter}$$

5.3) Reference Architectures

Reference architectures are high-level specifications representing common structures of architectures specific to a particular domain or platform. If they exist, they can play an essential role in the architectural definition of a software system. Theys often emerge in an abstracted and standardised form from successful architectures. Reference architectures define accepted structures and processes that help to build maintainable and interoperable systems. In our context, these architecture abstractions can be represented similar to architectural styles, i.e. at a meta-level in terms of architectural element types and their properties. What we add to our illustration of reference architectures is the behavioural perspective. We allow them to be described in terms of service interactions.

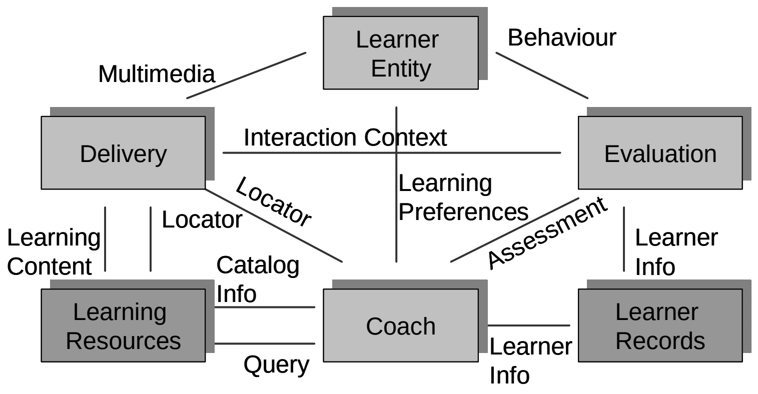


Figure 2. A Structural Overview of the LTSA Reference Architecture.

In the context of educational software systems, the Learning Technology Standard Architecture LTSA provides a service-oriented reference architecture (IEEE, 2001). It captures common structural and behavioural features of learning technology systems. We can describe IDLE's architectural characteristics using the LTSA.

- An abstract structural representation of the LTSA in a notation resembling UML class diagrams can be found in Fig. 2. In terms of the architecture ontology, we would describe as follows for the component definition.

$$\begin{aligned}&\mathit{LearnerEntity} \vee \mathit{Delivery} \vee \mathit{Evaluation} \vee \\ &\mathit{Learning\ Resources} \vee \mathit{Coach} \vee \mathit{LearnerRecords} \quad \subseteq \quad \mathit{Component}\end{aligned}$$

- The interaction behaviour (of the delivery parts) of the LTSA in terms of the interaction calculus can be described as follows.

```
LearnerEntity : =
        prefInfo = inv getPref (); inv setPref ( alter(prefInfo) );
        learnRes = inv multimedia ()
Coach :=
        repeat ( choice ( rcv getPref(); rep getPref(prefInfo),
                rcv setPref(prefInfo), uri = inv locator (resource) ) )
Delivery :=
        rcv locator(uri); learnRes = inv retrvRes (uri); rep multimedia (learnRes)
LearningResources : =
        rcv retrvRes(uri); rep retrvRes( retrieve(uri) )
```

An important goal of using a (service-oriented) reference architecture in our context is the identification of services in the original IDLE system:

- Some of the components are already services – an SQL execution element that is part of the lab resources and delivery subsystem is an example.
- Some components (implemented as Java objects) are not services – a feedback system is not encapsulated as a service; another example is the workspace function. We have decided to realise the workspace function, which could have been integrated into either learning resources or learner records, as a separate service in the re-engineered system.

5.4) Quality-Driven Architecture

Architectures and the architectural styles and reference architectures they are based on have a critical impact on the quality of a software system. The use of styles in architecture design implies certain properties of software systems, as these styles are abstractions of successfully implemented systems that are usually easy to understand, to manage, to maintain etc. While of course functional properties of services are vital, non-functional quality aspects ranging from availability, performance, and maintainability guarantees to costing aspects are equally important and need to be captured to clearly state the quality requirements. The reliability of a service-based system, the availability of services, and the individual service and overall system performance are often crucial. Links exist between functionally-oriented architecture models and quality properties of these systems (Garlan & Schmerl, 2006; Spitznagel & Garlan, 1998). A mere statement of required quality properties is therefore often not sufficient to actually guarantee these properties. We look at architectural styles and reference architectures to illustrate this point.

- A catalogue of architectural styles (Barrett et al., 2006) may be used by software architects to determine general patterns that would lead to architectures that exhibit some desired quality properties. Each of the styles in the catalog is associated with certain quality characteristics, which would be exhibited during the deployment and execution of system compositions. We return to this aspect later on in the context of patterns.
- Reference architectures are different in that large catalogues of these are usually not available. Reference architectures are often prescribed. The quality benefit is in terms of interoperability and reuse. Associated qualities, as for styles, are not the primary aim.

Quality-driven development requires quality attributes to be evaluated and confirmed. As architectural styles are often extensible and composable, the qualities of newly derived styles cannot always be taken for granted. Only through empirical evaluations can these expected qualities be confirmed. For instance, the Goal-Question-Metric (GQM) approach to quality goal evaluation (Basili et al., 1994), a method which allows metrics to be derived from abstract quality criteria, can support this quality evaluation endeavour. Implemented systems can be evaluated using the metrics derived from the quality goals via GQM, but this approach can also be used to address internal quality attributes. We will return to this aspect later on in the specific context of performance evaluation where a specific combination of goals and metrics is used.

6) Pattern-based Modelling, Architecture, and Development

The use of patterns in architecture design implies certain properties of services and systems, as these patterns are – similar to the higher-level styles and reference architectures – abstractions of existing system aspects that exhibit certain qualities. The implementation model of services in general and Web services in particular are based on the idea of service provider and service client being business partners. This constellation requires contracts to be set up, based on service-level agreements (SLAs). While of course functional properties of services are vital elements in these SLAs, non-functional aspects – ranging from availability, performance, and maintainability guarantees to costing aspects – are equally important and need to be captured in SLAs to clearly state the quality obligations and expectations of provider and client. Explicit models can support SLAs for service-based systems deployment.

Based on the requirements for quality-aware model-driven service engineering, this section covers specific techniques for model-driven development with pattern-based modelling using a UML-compliant technique based on functional and distribution patterns for service architectures at its core. The impact of the techniques on quality – of both the models and the final system – is highlighted. Layered pattern modelling with a strong emphasis on distribution patterns emerges as the crucial element.

We distinguish workflow patterns (van der Aalst et al., 2003) and architectural design patterns (Garlan & Schmerl, 2006). Workflow patterns relate to connector types that are used in the composition of services or components – they are actually provided as built-in operators of the calculus. Architectural design patterns are structural and behavioural constraints formulated on a number of components with particular roles. We join here design patterns (Gamma et al., 1995) and architectural patterns (Garlan & Schmerl, 2006) into one architectural design pattern concept.

6.1) Service Composition Meta-Model

Our core notation for service configurations as interaction processes and remote activations was based on a process calculus. A service process based on the orchestration, or composition, of individual services can be also formulated in terms of the UML activity diagram in order to use a common visual notation. Its (simplified) definition as a process language based on activity nodes and edges is presented in Fig. 3.

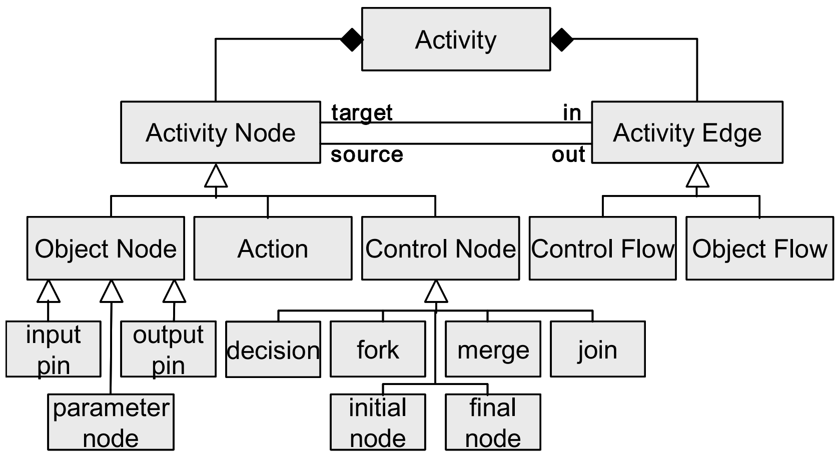


Figure 3. Meta-model for UML Activity Diagrams.

The process-centric architecture modelling notation and these UML activity diagrams are related. Here, the structural connectivity in terms of service composition operators such as sequence, choice or parallel composition can be represented in terms of UML using control flow nodes and edges such as decision, fork, merge and join. Thus, we combine textual architecture descriptions (typical for ADLs such as ACME) and visualisations in terms of UML (which is also used extensively for architecture modelling).

6.2) Layered Service Systems Modelling

A layered conceptual service architecture model that is tailored towards the needs of service and process-oriented platforms shall address the different levels of abstraction in service-based architectures:

- Architectural design patterns are medium-scale patterns – usually referred to as design patterns or architectural frameworks.
- Workflow patterns are process-oriented patterns that represent common business or workflow processes in an application domain.

Some of these patterns qualify as distribution patterns, which are platform-oriented patterns with certain quality characteristics attached. These patterns are extensions of architectural styles. While the styles were more structurally oriented, these patterns put an emphasis on interaction and processes. Similar to styles, their aim is reuse.

Design patterns are recognised as important building blocks in the development of software systems (Gamma et al., 1995). Their purpose is the identification of common structural and behavioural patterns in these systems. A rich set of design pattern has been described, which can be used to structure a software design at an intermediate level of abstraction. Design patterns in Web services architectures are discussed in (Topaloglu & Capilla, 2004). Usually, architectural patterns (such as client-server or model-view-controller) are distinguished from design patterns (such as factory, composite, or iterator). We see both forms of patterns as constraints on a system architecture, i.e. on services and on their patterns of interaction.

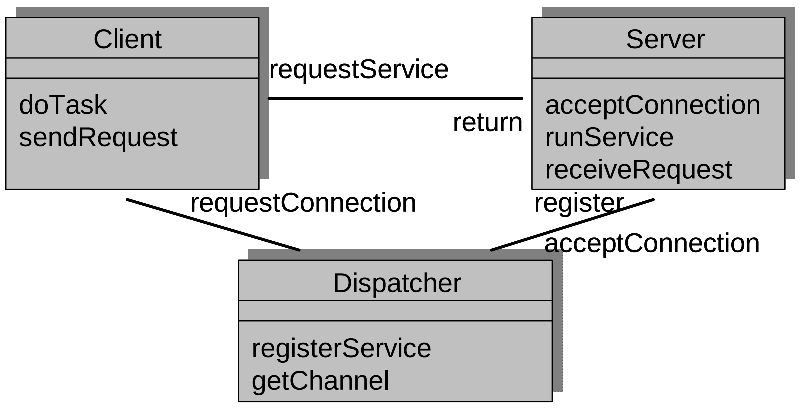


Figure 4. The Client-Dispatcher-Server Design Pattern.

An example of an architectural design pattern for service architectures is the client-dispatcher-server pattern (Topaloglu & Capilla, 2004), represented in Fig. 4. In IDLE, a learner requests content from a resources server. This involves the learner (client), a coach (dispatcher), and the resources and delivery subsystem (server). The pattern is not identical to the structure found in the IDLE system. However, the client-dispatcher-server pattern is simulated by the composite process

par (*LearnerEntity*,*Coach*,*Delivery*)

of an IDLE reformulation in LTSA terminology, i.e. the pattern is a good abstraction of IDLE functionality. Abstracted pattern definitions such as client-dispatcher-server can act as building blocks in higher-level architectural specifications. Patterns are defined as process expressions and made available as process abstractions.

Workflow patterns are small-scale process patterns. These are small compositions of basic activities. The multichoice pattern is an example:

mchoice (*Lecture*, *Tutorial*, *Lab*)

expresses that a selection of IDLE services *Lecture*, *Tutorial*, and *Lab* can be used concurrently. Workflow patterns and the problems they cause when implemented in Web services infrastructures are described in (van der Aalst et al., 2003). To identify these patterns is important since often not all of them are supported by the implementation languages. In this case, predefined architectural transformations can be reused in an MDD environment that generates executable processes automatically.

The IDLE storage/workspace feature can be integrated as a service:

```
WorkSpace :=
        choice ( repeat ( rcv retrieve (resId); inv provide (res) );
        repeat ( rcv store (resId, res) )  )
```

This explicit storage and workspace service would require for instance the services *LearnerEntity* and *Delivery* to be modified in their interaction patterns.

The central aim of patterns is reuse, which leads to improved quality as a result. The reuse of architectural design is one issue. For instance, the client-dispatcher-server pattern is a common pattern that divides functionality and achieves loose coupling, which is a way to improve maintainability and replaceability. Reuse in this case also means reuse of predefined transformations, as the mchoice-based workflow pattern shows.

6.3) Service Distribution and Topology Modelling

While patterns in general can influence some of a system's quality characteristics, such as understandability or maintainability, for service-centric software systems specific properties arising from the distributed and cross-organisational context are of central importance. The reliability of a system, the availability of services, and the individual service and overall system performance are often crucial. We introduce distribution patterns to provide a framework for higher levels of abstraction beyond the service process composition that addresses these quality aspects (Barrett et al., 2006). Links exist between functionally oriented models and quality properties of these systems. A mere statement of required quality properties is often not sufficient to actually guarantee these properties. We look at distribution properties of service-centric software systems to illustrate this point.

Distribution, i.e. the consideration of locations of services in a complex system, affects qualities of the software systems such as reliability, availability, and performance. We use the term service topology to refer to the modelling of service compositions as collaborating entities under explicit consideration of the distribution characteristics.

Based on experience in designing and implementing service-centric software systems, a number of standard architectural topologies have emerged for distributed, service-based systems (Thone et al., 2002; Skogan et al., 2004; Vasko & Duskar, 2004). They include centralised configurations such as the hub-and-hpoke or decentralised ones such as peer-to-peer architectures. These standard topologies, or configurations, can be abstracted into distribution patterns for the SOA platform. Distribution pattern modelling expresses how a composed system is to be deployed in a distributed environment (Skogan et al., 2004).

The goal is to enable the generation of architecturally flexible Web service compositions that have the desired quality characteristics and whose quality characteristics can be evaluated and altered at design level – we will address the latter aspect in the Section 7. 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 and also the previously discussed workflow and architectural design patterns can be expressed in the same way in our notation. All patterns refer to the high-level cooperation of components, termed collaboration. Workflows are compositional orchestrations, whereby the internal and external messages to and from services are modelled. Distribution patterns are, similar to design patterns, abstract compositional choreographies, where the focus is on external message flow between services. A choreography expresses how a system would be deployed in a distributed environment. We denote these compositions as distributed compositions by annotating the composition operators, e.g. for the hub-and-spoke, which is often called Centralised, the specification

$$\mathit{Centralised} = \textbf{\textit{d-par}}\ (\ \mathit{Hub}, \mathit{Spoke}_1, \ldots, \mathit{Spoke}_n\ )$$

denotes a parallel distribution.

This annotation of a composition is of importance if, for instance, an executable service process is generated. In an MDD solution to service engineering, the predominant execution language for service compositions is the process execution language WS-BPEL. In WS-BPEL, process partners would be configured as distributed services. Semantically, we follow the architecture description language ACME (Garlan & Schmerl, 2006) here and introduce a type language for architectural elements, i.e. processes are typed. The identifiers *Hub* and *Spoke* are actually service types that can be instantiated, for instance by *Coach* and *Learner* services, respectively. The distribution annotation denotes a distribution constraint that will have to be satisfied by a concrete implementation. Although code generation is an integral element of MDD, in the context of the Web service platform, WS-BPEL is the predominant executable service composition language. The transformation from the activity diagrams (or the interaction calculus) is uncritical; we therefore keep our focus on modelling activities. It is worth noting that architectural modelling constraints such as styles and reference architectures are meta-level constraints on architecture, i.e. need to be considered during architecture modelling and need to be satisfied by concrete architectures, but do not have to be considered for code generation.

Our framework comprises a catalogue of distribution patterns (Barrett et al., 2006). Each of the patterns in the catalogue is associated with certain internal and external quality characteristics. The patterns in the catalogue are split into three categories: core patterns, auxiliary patterns and complex patterns. Core patterns represent the simplest distribution patterns most commonly observed in Web service compositions. Auxiliary patterns are patterns which can be combined with core patterns to improve a given quality characteristic of a core pattern, the resultant pattern is a complex pattern. This catalogue assists software architects in choosing a distribution pattern for a given application context. The catalog categories are briefly outlined below:

- Core patterns are *Centralised* and *Decentralised*.
- An auxiliary pattern is the *Ring*.
- Complex patterns are *Hierarchical*, *Centralised Ring*, *Decentralised Ring*, and *Centralised and Decentralised Hybrid*.

We describe one pattern that can be applied in IDLE in detail to illustrate distribution patterns and their quality relevance. We consider here the hub-and-spoke pattern. This pattern abstracts a system that manages a composition from a single location, the hub, which is normally the participant initiating the composition. The composition controller (the hub) is usually remotely accessed by the participants (the spokes). This is the most popular and usually default distribution configuration for Web service compositions (van der Aalst et al., 2003). We specify *Hub* and *Spoke* as components, i.e. $Hub \subseteq Component$ and $Spoke \subseteq Component$. Suitable completeness and disjointness constraints would need to be added.

$$Hub = \exists\ hasPort\ .\ Input \quad \text{and} \quad Spoke = \exists\ hasPort\ .\ Output$$

explain that hubs receive incoming requests from spokes. Further constraints could limit the number of hubs to one, whereas spokes can be instantiated in any number. The dynamics can be specified as follows:

$$Centralised \quad = \quad \textbf{\textit{d-par}}\ (Hub, Spoke_1, \ldots, Spoke_n\ )$$

with

$$Hub \quad = \quad \textbf{\textit{repeat}}\ (\ \textbf{\textit{rcv}}\ invocation(\ldots);\ \textbf{\textit{rep}}\ reply(\ldots)\ )$$

$$Spoke_i \quad = \quad \textbf{\textit{inv}}\ result = invocation(\ldots)$$

Centralised is at activity level; *Hub* and *Spokes* are at interaction level. This could be specified in terms of UML Activity and Interaction Diagrams.

A sample application of the *Centralised* pattern in the IDLE context consists of the often widely distributed *Learner* client applications as the spokes and the centralised *Delivery* educational service provider as the hub.

The advantages of the *Centralised* or hub-and-spoke pattern in terms of quality aspects are:

- Composition is easily maintainable, as composition logic is all contained at a single participant, the central hub.
- Low deployment overhead as only the hub manages the composition.
- Composition can consume participant services that are externally controlled. Web service technology enables the reuse of existing services.
- The spokes require no modifications to take part in the composition. Web service technology enables interoperability.

The main disadvantages are:

- A single point of failure at the hub provides for poor reliability/availability.
- A communication bottleneck at the hub results in restricted scalability. SOAP messages have considerable overhead for message de-serialisation and serialisation.
- The high number of messages between hub and spokes is sub-optimal. SOAP messages are often verbose resulting in poor performance for Web services.
- Poor autonomy in that the input and output values of each participant can be read by the central hub.

All patterns have their advantages and disadvantages. The selection is determined by the context requirements. The hub-and-spoke pattern is typical for learning technology systems, for which maintainability and interoperability are central. Failure is not a highly critical problem and the number of users is predictable – which allows us to neglect two of the major disadvantages.

7) Performance-driven Modelling and Evaluation

We have looked at quality aspects of service-based software systems. Services as black-box entities limit this almost to the architectural level. In this section, however, we investigate a model-driven approach to the empirical quality evaluation of external quality properties. We focus on performance as one specific aspect.

We introduce an evaluation cycle, using a GQM approach for goal-to-measurement mapping. We extend service architecture modelling through an explicit and empirical way of dealing with quality. An empirical performance-oriented MDSE with instrumentation and measurement as example of one quality approach shall be presented.

Although model-based evaluation methods for performance exist, for example simulation and analytic methods, we choose an empirical approach here. Its benefits are accuracy and empirical validation. Our aim is to explore the potential of this technique under the given constraints given by the architecture-centricity of SOA and the black-box character of services to demonstrate the possibility of validated qualities.

7.1) Software Performance, Evaluation and Motivation

Performance is considered as the degree to which a software system or component meets its objectives for timeliness (Snodgrass, 1987). It can be evaluated with simulation techniques, with analytical modelling or using empirical methods (Lilja, 2000):

- Simulation is an imitation of a program execution. In simulations, only selected important parts of an execution are imitated. It is less expensive then building a full-scale software system for empirical evaluation. It is also flexible as changes can be dealt with easily if the simulation is derived automatically. However, simulation can suffer from a lack of accuracy.
- Analytical modelling is a technique where a system is mathematically described. Results of an analytical model can be less accurate than real-system measurements. However, analytical models are often easy to construct.
- Empirical evaluation is performed by measurements and metrics calculation. They provide the most accurate results as no abstractions are made.

We consider here model-based empirical performance evaluations in order to demonstrate the potential and limits of service-based quality evaluation through code-based measurement. Empirical evaluation can be seamlessly integrated into a model-driven development methodology, as we will demonstrate.

While code-level instrumentations and evaluation techniques for services exist, we feel that in model-driven software development, observations of behaviour and their evaluation should be done in the terms of modelling constructs. Instrumentation for observing software should be done in terms of modelling constructs in order to prevent the software architecture from having to deal with transformation details. A necessary part of empirical performance evaluation is the execution data collection, which is achieved through instrumentation. The next subsection gives an overview of the instrumentation problem.

7.2) Instrumentation, Measurement and Evaluation

In software engineering, instrumentation is the process of adding software probes to a program for observing system behaviour and evaluating system properties (Snodgrass, 1987). Software probes are pieces of code for collecting data about the software execution. Generally, there are two techniques for collecting data about a program execution, sampling and event tracing:

- Sampling is a technique where parts of a program are sampled during its execution in some time interval – an example is sampling the program stack to follow program execution. It is a statistical technique in which a representative sample of the data about the program during execution is taken. An advantage of this approach is that the impact on the performance of the program does not depend on the execution of the program. However, collected samples are different from run to run. The possibility that infrequent events are missed is another drawback.
- Event tracing is a process of generating traces of events in the software execution. A program trace is a dynamic list of events generated by the program as it executes (Lilja, 2000). A trace contains the time-ordered events and can be used to characterize the overall program behaviour. Problems that can be encountered with event tracing are system perturbations due the measurement and the amount of resources that tracing requires. Each newly added probe causes execution overhead (performance) and event traces require resources (memory).

Due to its greater reliability, we utilised event tracing. We represent traces in a relational format using temporal database concepts. Temporal databases are databases that support a notion of time (Snodgrass, 1987). In contrast to conventional, non-temporal databases, in which only facts are stored, each fact stored in a temporal database is associated with some time information. These facts can be related to a valid time dimension and to a transaction time dimension (Snodgrass, 1988). The valid time dimension is related to the time when the fact was true in reality. The transaction time dimension is related to the presence of the fact in the database. Temporal databases which store only facts about the past are called historical databases (Sarda, 1990). Historical databases define two kinds of relations, event and interval relations. Interval relations are used for storing facts which were true for some time interval. Event relations are used for storing facts which were true at some particular point of time.

7.3) Model-driven Service Development and Instrumentation

At present, most research in model-driven development is dedicated to simulation and performance prediction with mathematical analysis methods (Balsamo et al., 2004; Park & Kang, 2004). Nevertheless, predictions have to be validated when the software system is implemented and deployed. Validation should be based on modelling constructs as predictions are made according to them. Currently, timing behaviour is analysed based on source code constructs such as method execution time. In MDD, the level of abstraction is raised. Consequently, observations need to be based on modelling constructs such as states, activities, or methods.

We introduce an approach for the model-driven empirical performance evaluation of service-based software systems. We need to define a model-based language for service instrumentation. Instrumentation languages can enforce data collection in a relational format. We focus on compositions (orchestrations) of services to processes and address their performance behaviour. Our approach comprises:

- An instrumentation notation for service models that allows specific service model elements such as services or composition and flow operators to be annotated and marked as providing performance-relevant time information at execution time. We use UML activity diagrams to express service compositions and base our instrumentation language on this UML diagram format.
- Model-driven transformation techniques that generate executable code including the monitoring instructions necessary to record time information.
- A trace analysis query language that provides the ability to calculate performance metrics. The evaluation is based on the temporal databases theory (Zaniolo et al., 1997). The temporal databases theory relates facts stored in a relational format with time information. A relational program trace is a dynamic list of events and timing information generated by the program as it executes (Lilja, 2000).

The hypothesis of our approach is that the execution of a program, which is defined by modelling elements of a modelling language, can be characterised as either an event or an interval. The most important concepts in this basic package are thus interval trace and event trace. For instance, if an element of a modelling language models a part of the program execution which lasts for some time interval, it will be instrumented by a specialisation of the interval trace.

7.4) Model-based Instrumentation Language

The instrumentation technique is developed around an instrumentation language. This is going to be integrated with the service modelling language, i.e. is an extension of the UML activity diagrams that we have used to model service orchestrations. Both the orchestration language and the instrumentation language can be defined in terms of the Meta Object Facility (MOF) (OMG, 2006). Our instrumentation notation comprises of two parts.

- Firstly, a basic trace package that captures the notion of traces and its two variants, event and interval traces, and operations to capture these traces (Fig. 5).
- Secondly, the instrumentation of activity diagrams using the MOF profiles extension mechanism (Fig. 6).

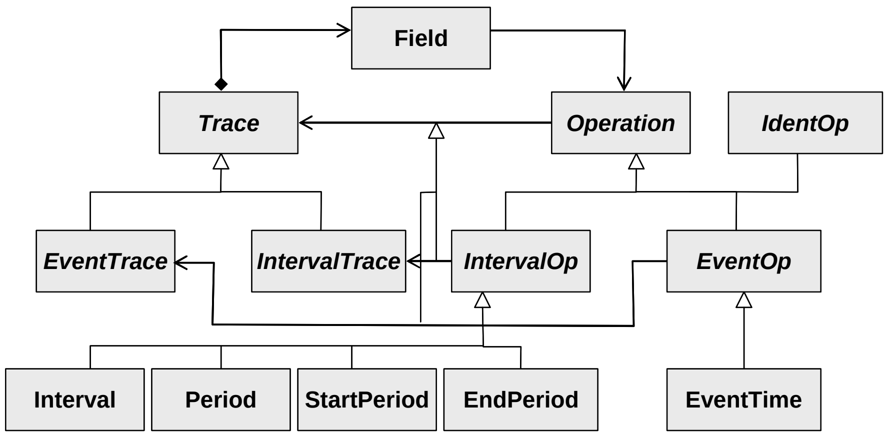


Figure 5. Basic Trace Package.

The basic trace package reflects the required time dimensions and the recording concepts. The activity diagram instrumentation utilises these concepts. This separation allows the basic instrumentation principles to be reused across a range of problem-specific or even model-specific applications. In the given instrumentation, actions such as the central elements of activity diagrams and all six control nodes are annotated. The execution of actions, which represent services at the model level, takes some time, i.e. an interval trace should be recorded at performance evaluation or execution time. We assume control flow decisions such as the start and end of the overall process, choices or mergers as instantaneous events.

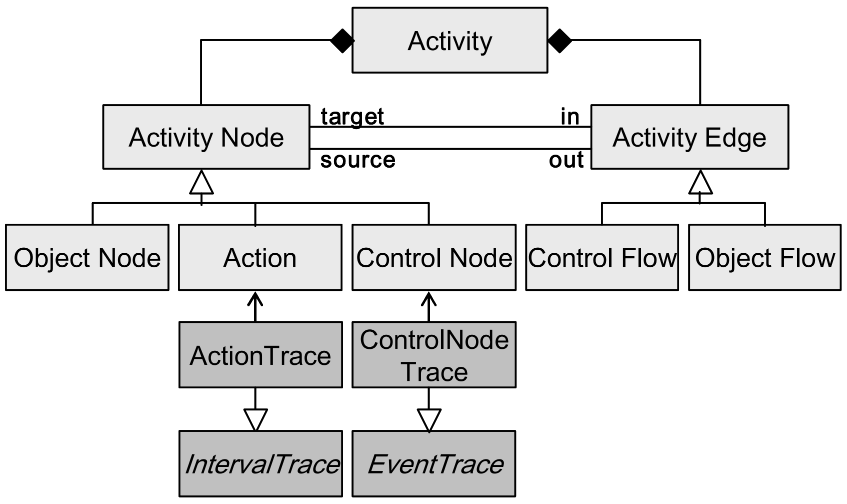
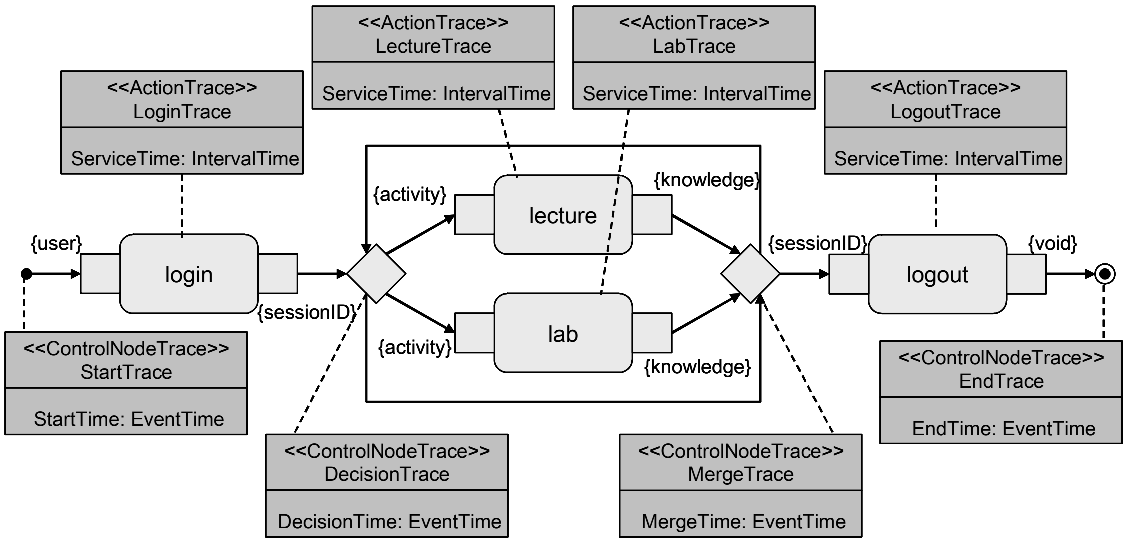


Figure 6. UML Activity Diagram Instrumentation.

7.5) Instrumentation Application

The application of the instrumented activity diagram is illustrated in Fig. 7.Two types of model elements – actions such as login or transfer and control nodes such as the start or the first decision point – are instrumented. An interval consisting of begin and end time of the service executions that implement the actions are recorded as a consequence of this instrumentation. Events, i.e. individual time stamps, are recorded for the control nodes.



<<ActionTrace>> LectureTrace
ServiceTime: IntervalTime
<<ActionTrace>> LabTrace
ServiceTime: IntervalTime
<<ActionTrace>> LoginTrace
ServiceTime: IntervalTime
<<ActionTrace>> LogoutTrace
ServiceTime: IntervalTime
{activity}
lecture
{knowledge}
{user}
login
{sessionID}
logout
{void}
{sessionID}
lab
{activity}
{knowledge}
<<ControlNodeTrace>> StartTrace
StartTime: EventTime
<<ControlNodeTrace>> EndTrace
EndTime: EventTime
<<ControlNodeTrace>> DecisionTrace
DecisionTime: EventTime
<<ControlNodeTrace>> MergeTrace
MergeTime: EventTime

Figure 7. Application of the Instrumentation to the IDLE Activity Selection Process.

For the modeller and service architect, it is import to find an adequate instrumentation that provides answers to the relevant performance questions. For instance, in a particular situation only the (average or maximum) response times of particular services, such as the lecture and lab activity services, are of interest. Then, the instrumentation needs to reflect these requirements.

While we consider this instrumentation of actions and control nodes to be the standard case, the approach is actually flexible enough to accommodate context-specific customisations. Some of the control nodes could be excluded or other modelling elements could be additionally included. This is only limited by the extent to which the transformation and code generation supports the different model element instrumentations. Some guidance could be provided by disabling the instrumentation of elements that are difficult to implement or whose analysis would not provide useful performance information.

7.6) Performance Monitoring

The actual implementation of the instrumentation is critical insofar it should be, firstly, easy to realise and, secondly, implemented without significant overhead. Aspects and interception techniques can be used to implement the instrumentation and data collection.

In the services context, often the problem arises that the addition of probes into the service implementation is not possible due to the nature of services as black-box software components. We therefore distinguish two scenarios:

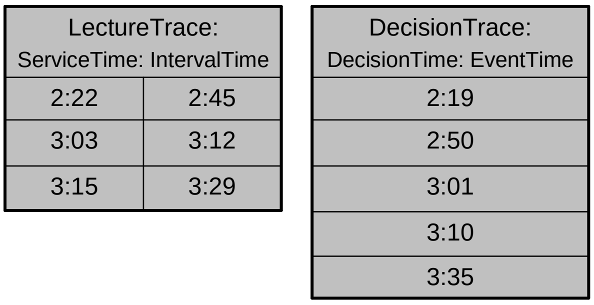
- Controlled environments that allow access to code. Aspect Oriented Programming (AOP) is a programming approach, suitable for the controlled approach, which can be used for transparent software instrumentation. The source code of the software is here not mixed with probes (Debusmann & Geihs, 2003). Marenholz et al. (2002) use AspectC++ for the instrumentation of operating systems for debugging, profiling/measurement, and runtime surveillance/monitoring. AOP is a technique that enables the separation of instrumentation from the development of the core software functionality. With AspectJ and AspectC++, instrumentation can also be done by adding a transparent software layer to the application for collecting execution data.
- Open environments in which services are black-box components. For a transparent instrumentation of service systems, interceptors can be used. Interceptors are similar to AOP and can intercept method invocations to transparently instrument a program (Klar et al., 1991; Debusmann et al., 2002, Yeung et al., 2004). For instance, software probes are predefined and placed in stubs and skeletons during an interface description compilation (Debusmann et al., 2002). The measurement probes can be turned on and off at runtime. Diaconescu et al. (2004) introduce an approach where a transparent proxy layer for data collection is automatically generated at deployment time.

The JBoss Application Server, for instance, enables the transparent aspect-oriented addition of functionality. Its AOP features allow the interception of events and addition of trigger functionality based on those events. AOP, interceptors, and bytecode and platform instrumentation are approaches that enable the collection of data without influencing the functionality code. We can employ these ideas to collect data about the software execution at the model level as a separate concern.

The first step, however, is the generation of executable and instrumented code. Activity diagrams that model service orchestrations can be converted into executable WS-BPEL Web services processes if invocation information such as service location is added to the abstract service process description:

- AOP concepts are used to generate the instrumented executable service code.
- Interception mechanisms are used to add the instrumentation and data collection.

We suggest using the ATL transformation language and tool from the ATLAS project to transform activity diagrams into executable code. The instrumentation includes monitoring and data collection functionality. Data is stored in a temporal or historical database – or by using the time extensions of traditional relational databases. Fig. 8 shows a sample recording based on the instrumentation defined in Fig. 7.



LectureTrace: ServiceTime: IntervalTime	
2:22	2:45
3:03	3:12
3:15	3:29

DecisionTrace: DecisionTime: EventTime
2:19
2:50
3:01
3:10
3:35

Figure 8. Collected Data for Learning Activity Process Instrumentation.

7.7) Performance Evaluation

Performance-relevant information needs to be extracted from the basic times stored in the database in order to allow a software architect to assess the overall performance of individual services and also orchestrated service processes. Temporal and historical databases are usually extensions of traditional relational databases. SQL is therefore available as a query language to retrieve and aggregate information based on the recorded event and interval times. We argue that SQL is actually sufficient as a query language to formulate the relevant performance assessment queries. More advanced solutions like data warehouses with their extended evaluation support are not required for typical performance assessments. Two central performance assessments issues are:

- Response time assessment: response times of activities are usually recorded as intervals. The SQL aggregate functions, such as average (AVG) or maximum (MAX), provide the relevant answers.
- Frequency and distribution of invocations: the distribution of invocations (workload) between the individual services can be determined based on the calculation of ratios between total numbers of invocations.

The database representation directly reflects the modelling layer, as the representation is generated from the model instrumentation. The queries are consequently formulated in terms of relevant model elements – which is one of the central objectives of our model-driven performance evaluation approach.

The average response time for service *Lecture* can be determined as follows:

```
SELECT AVG (ServiceTime)
FROM   LectureTrace
```

The determination of the maximum time would be formulated in a similar way. In the context of SOA, where individual services are often provided by external organisations, this information is often part of contracts and service-level agreements.

The proportion of *Lecture* invocations in relation to all user selections (decisions) can also be formulated:

> *SELECT COUNT* (*ServiceTime*) / *COUNT* (*DecisionTime*)
> *FROM LectureTrace*, *DecisionTrace*

This allows a software architect to judge the frequency of individual service activations in typical application scenarios.

8) Conclusions

Since the service platform is based on a business model involving service-level agreements between providers and users, more than the service's functionality needs to be agreed upon. Internal qualities such as maintainability, but also external and observable quality attributes such as performance or security are of central importance for clients and providers.

Quality assurance is, however, a challenge as a consequence of the black-box character of services, at least from the client perspective. Service-oriented architecture (SOA) is an integration approach and consequently architecture-centric. Modelling in general and quality-aware modelling techniques in particular need to provide tailored service- and architecture-centric solutions.

In the context of this arising need to address quality for services in a model-driven service engineering discipline, we have investigated the crucial quality aspects and techniques that can be employed to actually address these through modelling activities at the service architecture level.

- Abstract architectural constraints in the form of styles and reference architectures guide architectural modelling towards interoperability and maintainability.
- Pattern-based modelling of service architectures is a further step towards reuse and maintainability.
- Empirical evaluation techniques, for instance for performance, complement the previous focus in internal quality attributes by external quality considerations.

The focus on one specific aspect, performance, in the evaluation context shows that although architectural approaches allow referring to and addressing a variety of qualities as far as modelling is concerned, more specific techniques are needed to evaluate these qualities. Performance, security, or maintainability require different approaches to be integrated in a quality-aware service engineering discipline.

As such a service engineering discipline is only emerging, we have discussed our findings as part of a roadmap to a comprehensive quality-aware service engineering approach. Although quality assurance is difficult to achieve due to the character of services, with the architectural modelling focus and the empirical evaluation we have suggested two ways of dealing with internal and external quality attributes of service-based software systems, respectively.

9) Future Research Directions

Model-driven development for service-oriented architecture (SOA) might initially seem easy to tackle since SOA is an integration approach at the software and system architecture level. Abstraction is already given and code generation seems easy due to the service platform principles and predominate execution languages. Difficulties, however, arise as this simplification becomes a problem if quality assurance is an issue.

SOA also creates other still unsolved problems for model-driven development (MDD).

- Modelling of service-based systems itself is intrinsically different from traditional software modelling. Service-based systems are process-centric compositions. Adequate modelling and architecture notations need to be provided for an MDD approach.
- Due to the heterogeneity of SOA targets, information integration is another integration dimension in addition to service-level integration. The SOA community already investigates semantically enhanced models and descriptions in the form of ontologies – for both information and service aspects. Besides consistency across applications, a higher degree of automation would be enabled through semantic enhancements. A similar trend can be observed in the context of MDD, where currently an ontology definition metamodel to support semantic modelling is being standardised by the OMG.

Quality-aware model-driven service architecture needs to be linked to the service platform in order to deliver validated quality guarantees. We have only introduced platform instrumentation and interception mechanisms briefly here, but an in-depth investigation would allow the trade-off, for instance between accuracy and overhead, to be discussed in detail.

More within the concrete framework of our approach than the semantic enhancements, a number of issues have remained unaddressed. We have only looked at performance as one of the highly important external, observable qualities of a service-based system. Due to the distribution of service systems and the openness of the service platform in terms of communications infrastructure, security is another important quality. Corresponding modelling concepts for security mechanisms ranging from encryption to access control to trust control need to be integrated. This is at least an equally complex endeavour to our solution of model instrumentation, monitoring and evaluation for performance considerations.

The idea of semantically enhanced models can lead to a solution in the security and trust context. Certified semantic descriptions of functional and quality properties of service and system, formalised in terms of ontologies, can support service-level agreements, even automated composition of services from different providers. This, however, is a vision that is far from being investigated sufficiently.

References

van der Aalst, W.M.P., Kiepuszewski, B., ter Hofstede, A.H.M. and Barros, A.P. (2003). Workflow Patterns. *Distributed and Parallel Databases*, *14*, 5–51.

Abowd, G., Allen, R. and Garlan, D. (1995). Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, *4*(4), 319–364.

Allen, R. and Garlan, D. (1997). A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3), 213–249.

Alonso, G., Casati, F., Kuno, H. and Machiraju, V. (2004). *Web Services – Concepts, Architectures and Applications*. London, UK: Springer-Verlag.

Baresi, L., Heckel, R., Thöne, S. and Varro, D. (2004). Style-based refinement of dynamic software architectures. *Proc. 4th Working IEEE/IFIP Conference on Software Architecture WICSA* (pp. 155–164).

Barrett, R., Patcas, L.M., Murphy, J. and Pahl, C. (2006). Model Driven Distribution Pattern Design for Dynamic Web Service Compositions. *International Conference on Web Engineering ICWE'06* (pp. 129-136).

Basili, V., Caldiera, G., and Rombach, D. (1994). The Goal/Question/Metric approach. *Encyclopedia of Software Engineering*, Volume I, 528–532. Los Alamitos, CA: Wiley.

Bass, L., Clements, P. and Kazman, R. (2003). *Software Architecture in Practice*. SEI Series in Software Engineering. Boston, MA: Addison-Wesley.

Cortellessa, V., Di Marco, A. and Inverardi, P. (2006). Software performance model-driven architecture. *Proc. ACM Symposium on Applied Computing SAC'06* (pp. 1218–1223).

Cuesta, C. E., del Pilar Romay, M., de la Fuente, P. and Barrio-Solorzano, M. (2005). Architectural Aspects of Architectural Aspects. *Proc. 2nd European Workshop on Software Architecture EWSA 2005.* Springer LNCS 3047.

Debusmann, M. and Geihs, K. (2003). Efficient and Transparent Instrumentation of Application Components using an Aspect-oriented Approach. *Proc. IFIP/IEEE*

Workshop on Distributed Systems: Operations and Management DSOM 2003 (pp. 209–220). Springer LNCS 2867.

Debusmann, M., Schmid, M. and Kroeger, R. (2002). Measuring End-to-End Performance of CORBA Applications using a Generic Instrumentation Approach. *Proc. 7th Int. Symp. on Computers and Communications ISCC '02* (pp. 181–186).

Diaconescu, A., Mos, A. and Murphy, J. (2004). Automatic Performance Management in Component Based Systems. *Proc. 1st Int. Conf. on Autonomous Computing ICAC'04* (pp. 214–221).

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Design*. Boston, MA: Addison Wesley.

Garlan, D. and Schmerl, B. (2006). Architecture-driven modelling and analysis. *Proc. 11th Australian Workshop on Safety Related Programmable Systems SCS'06*, volume 69 of Conferences in Research and Practice in Information Technology.

Giesecke, S. (2006). A Method for Integrating Enterprise Information Systems based on Middleware Styles. *Proc. International Conference on Enterprise Information Systems ICEIS'06*, Doctoral Symposium (pp. 24–37).

IEEE Learning Technology Standards Committee LTSC (2001). *IEEE P1484.1/D8. Draft Standard for Learning Technology - Learning Technology Systems Architecture LTSA*. IEEE Computer Society.

ISO/IEC. *ISO 9126 Software Engineering – Product Quality – Part 1: Quality Model*. Published Standard.

Kazman, R., Carriere, S.J. and Woods, S.G. (2000). Toward a Discipline of Scenario-based Architectural Evolution. *Annals of Software Engineering*, *9*(1-4), 5–33.

Klar, V., Quick, A. and Soetz, F. (1991). Tools for a Model–driven Instrumentation for Monitoring. *Proc. 5th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation* (pp. 165–180).

Lilja, D. J. (2000). *Measuring Computer Performance: A Practitioner's Guide*. Cambridge, UK: Cambridge University Press.

Mahrenholz, D., Spinczyk, O. and Schroeder-Preikschat, W. (2002). Program Instrumentation for Debugging and Monitoring with AspectC++. *Proc. 5th Int. Symp. on Object-Oriented Real-Time Distributed Computing ISORC'02* (pp. 249–256).

Medvidovic, N. and Taylor, R.N. (2000). A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, *26*(1), 70-93.

Object Management Group (2003). *MDA Model-Driven Architecture Guide V1.0.1*. OMG.

Object Management Group (2004). *MOF 2.0, OMG document ptc/04-10-15*. web: http://www.omg.org/cgibin/apps/doc?ptc/04-10-14.pdf.

Oquendo, F., Warboys, B.C., Morrison, R., Dindeleux, R., Gallo, F., Garavel, H. and Occhipinti, C. (2005). ArchWARE: Architecting Evolvable Software. *Proc. 2nd European Workshop on Software Architecture EWSA 2005*. Springer LNCS 3047.

Pahl, C. (2002). A Formal Composition and Interaction Model for a Web Component Platform. *ICALP'2002 Workshop on Formal Methods and Component Interaction*. Malaga, Spain. Elsevier. Electronic Notes in Theoretical Computer Science.

Pahl, C., Barrett, R. and Kenny, C. (2004). Supporting Active Database Learning and Training through Interactive Multimedia. *Proc. Intl. Conf. on Innovation and Technology in Computer Science Education ITiCSE'04*.

Pahl, C. (2005). Layered Ontological Modelling for Web Service-oriented Model-Driven Architecture. *European Conference on Model-Driven Architecture ECMDA'2005*. Springer-Verlag, LNCS Series.

Pahl, C. and Zhu, Y. (2006). Semantical Framework for the Orchestration and Choreography of Web Services. *International Workshop on Web Languages and Formal Methods WLFM'05*. Newcastle upon Tyne, UK. Elsevier ENTCS Series.

Park, D. and Kang, S. (2004). Design phase analysis of software performance using aspect-oriented programming. *Proc. 5th Aspect-Oriented Modeling Workshop*, UML'2004.

Plasil, F. and Visnovsky, S. (2002). Behavior Protocols for Software Components. *ACM Transactions on Software Engineering*, *28*(11), 1056-1075.

Sangiorgi, D. and Walker, D. (2001). *The π-calculus – A Theory of Mobile Processes*. Cambridge, UK: Cambridge University Press.

Sarda, N. (1990). Extensions to SQL for Historical Databases. *IEEE Transactions on Knowledge and Data Engineering*, *2*(2), 220–230.

Selic, B. (2003). The Pragmatics of Model-Driven Development. *IEEE Software*, *20*(5), 19–25.

Skogan, D., Grønmo, R. and Solheim I. (2004). Web Service Composition in UML. *Proc. 8th International IEEE Enterprise Distributed Object Computing Conference EDOC'2004* (pp. 47-57).

Snodgrass, R. (1987). The temporal query language tquel. *ACM Trans. Database Syst.*, *12*(2), 247–298.

Snodgrass, R. (1988). A Relational Approach to Monitoring Complex Systems. *ACM Transactions on Computer Systems*, *6*(2), 157–196.

Spitznagel, B. and Garlan, D. (1998). Architecture-based performance analysis. *Proc. Conference on Software Engineering and Knowledge Engineering SEKE'98*.

Thöne, S., Depke, R. and Engels, G. (2002). Process-Oriented, Flexible Composition of Web Services with UML. *Proc. Joint Workshop on Conceptual Modeling Approaches for e-Business eCOMO 2002*.

Topaloglu, N.Y. and Capilla, R. (2004). Modeling the Variability of Web Services from a Pattern Point of View. *Proc. European Conf. on Web Services ECOWS'04* (pp. 128–138). Springer LNCS 3250.

Vasko, M. and Duskar, S. (2004). An Analysis ofWeb Services Flow Patterns in Collaxa. *Proc. European Conf. on Web Services ECOWS'04* (pp. 1–14). Springer LNCS 3250.

Weber, H. (2005). From Programme Engineering to Software Engineering . *Proc. Theory and Practice of Software Development TAPSOFT'2005*. (invited talk).

Yeung, K., Kelly, P.H.J. and Bennett, S. (2004). Dynamic Instrumentation for Java Using a Virtual JVM. *Performance Analysis and Grid Computing*, 175–187.

Zaniolo,C., Ceri, S., Faloutsos, C., Snodgrass, Subrahmanian, V. S. and Zicari, R. (1997). *Advanced Database Systems*. San Fransisco, CA: Morgan Kaufmann Publishers.