



Engineering Virtual Domain-Specific Service Platforms

Specific Targeted Research Project: FP7-ICT-2009-5 / 257483

View-based Design Time and Runtime Architecture for Tailoring VSPs

Abstract

In INDENICA, the notion of Virtual Service Platforms (VSP) shall be leveraged to protect the investments into service-based applications against potential external negative influences and threats such as the heterogeneity of the involving service platforms, service discontinuation, service evolutions, etc. In this report, we present a view-based design time and runtime architecture for developing and tailoring VSPs. The view-based approach supports each stakeholder to work with the most appropriate views for his current work tasks. The Model-Driven Software Development paradigm shall be exploited in order to separate the levels of abstraction. That is, high-level, technology- and domain- independent concepts and elements are separated from the technology- and domain-specific ones. This way, we can better support stakeholders in formulating and tailoring a particular VSP that integrates different service platforms. We also present the plans for collaborating and integrating the view-based architecture with other components of INDENICA.

Document ID:	INDENICA – D3.1
Deliverable Number:	D3.1
Work Package:	3
Type:	Deliverable
Dissemination Level:	PU
Status:	Final
Version:	1.01
Date:	2011-10-18
Contributing partners	UNIVIE, PDM, SAP, SIE, TUV, SUH, TEL

Project Start Date: October 1st 2010, Duration: 36 months

Version History

0.1	01. Jun 2011	Initial version
0.2	15. Jun 2011	Update Section 2
0.3	20. Jun 2011	Add figures of view models to Section 2
0.4	15. Jul 2011	Update the content w.r.t. the plan in Hildesheim meeting
0.5	22. Jul 2011	Update figures, add motivations in Section 1
0.6	08. Aug 2011	Revise content, add more figures
0.7	12. Aug 2011	Add the overall architecture
0.8	18. Aug 2011	Update the content in Section 2.4 and 3.2
0.9	26. Aug 2011	Update with TUV's contribution in Section 5.6 and Figure 11
0.10	29. Aug 2011	Update the content in Section 3.3
0.11	05. Sep 2011	Update Chapter 2. Update chapter 5.4.
0.12	15. Sep 2011	Merge partners' contributions and revise the content
0.13	22. Sep 2011	Consolidate contributions and address reviewers' comments
0.14	28. Sep 2011	Update Section 5.7
1.0	29. Sep 2011	Finalise the deliverable
1.01	18. Oct 2011	Corrected Typo

Document Properties

The spell checking language for this document is set to UK English.

Table of Contents

Table of Contents	3
1 Introduction.....	6
1.1 Motivations	6
1.2 Objectives	6
1.3 Relationships with other INDENICA components	7
2 View-based Design Time and Runtime Architecture for Tailoring VSPs	8
2.1 INDENICA Virtual Service Platform	8
2.2 Overall Platform Architecture View	8
2.3 Overall View-based Architecture	10
2.4 Core Model	12
2.5 Service Component View.....	12
2.6 The Service Deployment View	15
3 Runtime Realization and Code Generation.....	17
3.1 Extension mechanisms	17
3.2 Low-level, technology specific views	18
3.2.1 Introduction to Service Component Architecture (SCA)	18
3.2.2 The Low-level Service Component View for SCA.....	19
3.2.3 Refining high-level to low-level Service Component View.....	21
3.3 Runtime View.....	23
3.4 Code Generation	25
4 Roles of various stakeholders	29
5 Integration with other INDENICA components.....	30
5.1 Integration of View Models with Goal Models.....	30
5.2 Integration of View Models with Architectural Decisions.....	31
5.3 Integration of View Models with Variability Modelling	31
5.4 Integration of View Models with Enterprise Architecture Management	32
5.5 Integration of Adaptation and Monitoring.....	35
5.6 Integration of Deployment	38
5.7 Integration of Platform Service.....	39
6 Conclusion	41
Table of Figures	43

References 44

Appendix 46

Abbreviations and acronyms

AD	Architectural Decision
AK	Architectural Knowledge
EAM	Enterprise Architecture Management
ERP	Enterprise Resource Planning
DSL	Domain-Specific Language
MDSD	Model-Driven Software Development
QoS	Quality of Service
SCA	Service Component Architecture
VSP	Virtual Service Platform
WMS	Warehouse Management System
WSDL	Web Service Definition Language

1 Introduction

1.1 Motivations

Today's service-oriented approaches bring huge impact on the market that leads to a plethora of service platforms. Examples include service platforms providing services for enterprise resource planning (ERP), data storage, mobile communication, etc. That naturally leads to a considerable amount of challenges in developing a service-based application that typically integrates functions provided by different platforms.

First, there is a high degree of heterogeneity of the involving service platforms leading to various fragmentations such as quality fragmentation, interface fragmentation, and technical fragmentation [DoW]. *Second*, it becomes too complex to develop and maintain the service-based application using a single representation, from a single perspective, by a single stakeholder. As a consequence, it poses several business and technical risks such as service discontinuity, service evolutions, and so on.

In INDENICA, the notion of Virtual Service Platform shall be leveraged to shield the investments into service-based applications against the aforementioned potential external negative influences and threats. In particular, we aim at providing unified development interfaces, techniques, and tools for reconciling the heterogeneity of service platforms. A view-based approach shall be developed to support each stakeholder to work with the most appropriate views for his current work tasks. Moreover, the Model-Driven Software Development (MDSD) paradigm shall also be leveraged in order to separate the levels of abstraction. That is, high-level, technology- and domain- independent concepts and elements are separated from the technology- and domain-specific ones. This way, we can better support stakeholders in formulating and tailoring a particular VSP that integrates different service platforms.

1.2 Objectives

This deliverable is part of the WP3. WP3 aims at delivering an architecture, component, and tools for implementing and tailoring of service platforms. The major objectives of WP3 are including:

- Development of a view-based design time architecture for tailoring Virtual Service Platforms
- Development of a runtime architecture for tailoring Virtual Service Platforms and their governance
- Development of tools and generator templates which allow the tailoring and adaptation of Service Platforms
- Development of tools for Service Platform Engineering

The deliverable D3.1 shall report the results of task T3.1 in WP3. Task 3.1 aims at designing the *view-based design time architecture for tailoring virtual service platforms*. The major achievement is an architecture based on the notion of architectural views and the model-driven software development paradigm to, on the one hand, separate different dimensions of service platform variability (identified in

WP2), and, on the other hand, enable the ability to support various stakeholders' perspectives and enhance the automation in developing virtual service platforms.

1.3 Relationships with other INDENICA components

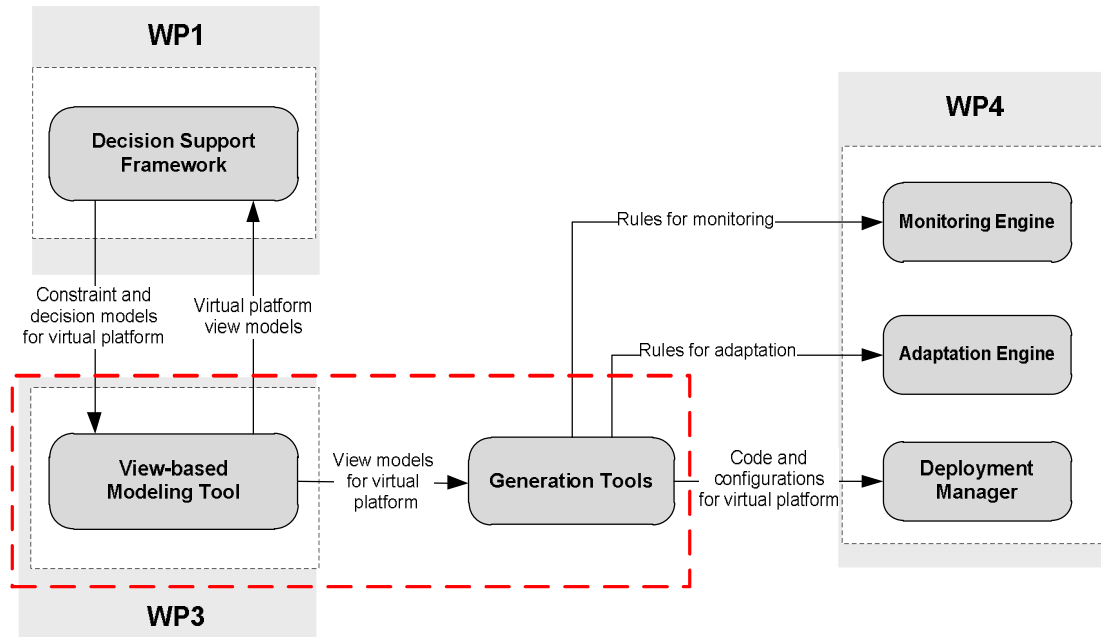


Figure 1 Relationships with other INDENICA components

The major components presented in this report are the *View-based Modelling Tool* and the *Generation Tools* (see Figure 1). The View-based Modelling Tool provides stakeholders with different view models from different perspectives of the virtual service platform under consideration. It aims at reducing the complexity of the virtual service platform models as well as enhancing the ability to tailor the platform models to different stakeholders' domains of interest. Based on the view models for a virtual service platform created using the View-based Modelling Tool, the Generation Tools can offer the stakeholders systematic ways to generate code such as service definitions, skeletons for service implementation, deployment configurations, monitoring directives, and so on.

The development of view models using the View-based Modelling Tool shall take into account the design decisions and constraints stemmed from the requirements of the resulting virtual service platform. These decisions and constraints are the outcomes of the Decision Support Framework developed in WP1. In addition, mechanisms provided by the Generation Tools shall support different kinds of generated artefacts that are inputs for the Deployment Manager, Monitoring Engine, and Adaptation Engine developed in WP4. The bespoke relationships shall be explained further in Section 5.

2 View-based Design Time and Runtime Architecture for Tailoring VSPs

In this section, we present our view-based design time runtime architecture and for developing and tailoring virtual service platforms.

2.1 INDENICA Virtual Service Platform

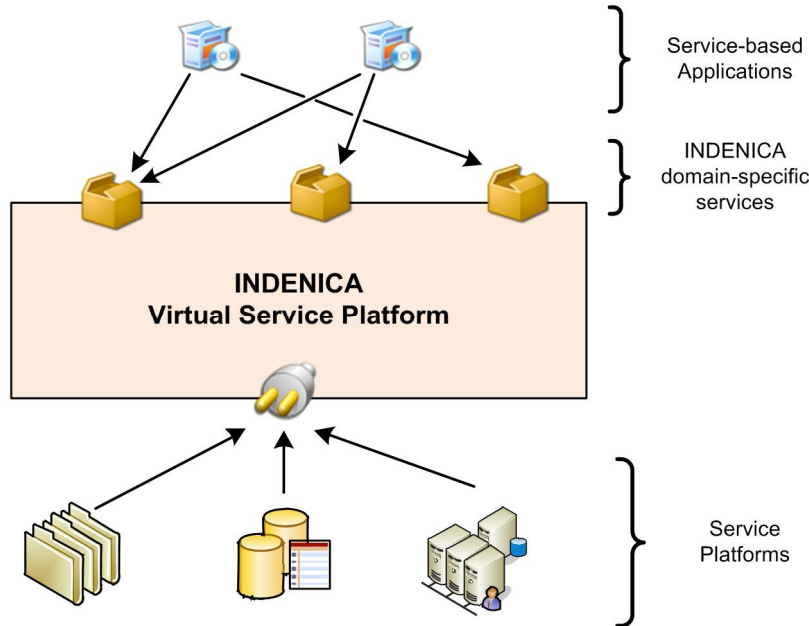


Figure 2 Overview of an INDENICA Virtual Service Platform

The main focus of INDENICA is to address the heterogeneity and integration of multiple service platforms used by a certain service-based application in a systematic way. Such heterogeneity is leading to a strong fragmentation of service platforms along various dimensions such as quality fragmentation, interface fragmentation, and technical fragmentation.

As a result, companies are forced to create virtual service platforms (VSP) in order to, on the one hand, shield their applications from such fragmentations, and on the other hand, reduce the risks such as service discontinuations, payment model changing, etc. caused by the dependencies between service providers and consumers. According to the specific requirements of the service-based applications atop, the underlying virtual service platforms can be substantially different.

To the best of our knowledge, there is a lack of existing approach, technique, or framework that supports stakeholders to efficiently develop such VSPs. In the next sections, we present a view-based, model-driven approach for enabling the systematic development of VSPs, enhancing the reusability of VSP development artefacts, and tailoring to various stakeholders' perspectives.

2.2 Overall Platform Architecture View

Especially during the development of a VSP an all-embracing architectural view is indispensable to handle the complex task of planning, tailoring and managing the

service platform. The view introduced in this section is derived from the Siemens Enterprise Architecture Management (EAM) methodology and offers an overview about the overall platform architecture, ranging from the actual business functions over integration and infrastructure layer to supporting guidelines and processes. This view furthermore allows the distinction between pure business application components and more generic application components (e.g. logging or monitoring) that are shared between the business application components.

While business functions and can be refined to high-level and low-level services using the service architecture view, the platform architecture view deals with global aspects of the platform development and provides insight to the overall target architecture at an abstract level. It is therefore possible to define the infrastructure components that will serve as base and runtime for the business components and document decisions like the choice of the application server. Furthermore, non-technical cross-cutting concerns like quality management or SoA governance can be addressed using this view.

The model itself is flexible and can easily be adapted to the needs of the domain as it only defines the architectural layers without specifying any concrete application or infrastructure components. All components and functions mentioned in Figure 3 are just placeholders used for illustrating the intended usage of the model. If, for instance, no rules engine is needed by the service platform, it can be left out.

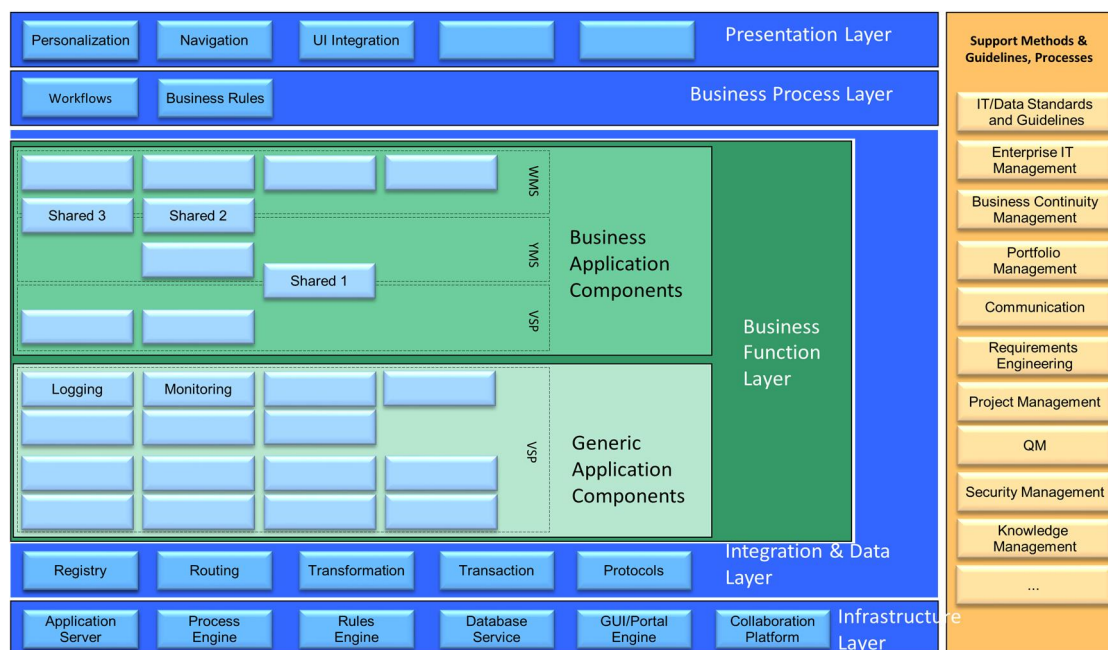


Figure 3: the technical architecture overview diagram from the EAM methodology with sample components in the respective layers

While the categorization of components that are not directly related to business cases is rather simple, deciding whether a component is generic or not may be more elaborate. One possible approach is the creation of a usage matrix in order to retrieve qualitative results on redundant functionality that is scattered over various components. This information may then be used to identify candidates for shared components. Afterwards the business value of each candidate can be determined by

examining factors like estimated cost or development time. All components whose value-cost ratio exceeds a pre-defined threshold are considered to be generic components. The resulting list of shared components can be further prioritized and allows for the creation of a roadmap for the realization of the service platform.

The following sections will focus on the refinement of the business functions and their mapping to actual services on different abstraction levels.

2.3 Overall View-based Architecture

An important element of service platforms is the concept of *service*. A service embraces an essential computational unit having functionality exposes via standardized interfaces (aka contracts). For instance, a Web service's interface can be defined using WSDL [W3C] whilst an OSGi service's interface can be described using the OSGi manifest description [OSGi]. Service interface descriptions might comprise further information such as communication protocols, security policies, and so forth.

A typical service platform often provides platform-specific services working on top of a number of infrastructure services. Services provided by multiple platforms can be combined in order to fulfil a certain business goal, for example, handling a customer order, booking a travel itinerary, etc. Service implementations and service-based applications need to be deployed to execute in an adequate hosting platform [MM2004, DS2005]. A virtual service platform (VSP) is a special kind of service platform aiming at shielding the heterogeneity of the underlying service platforms away and providing appropriate abstract layers to the service-based application developers (see Figure 2).

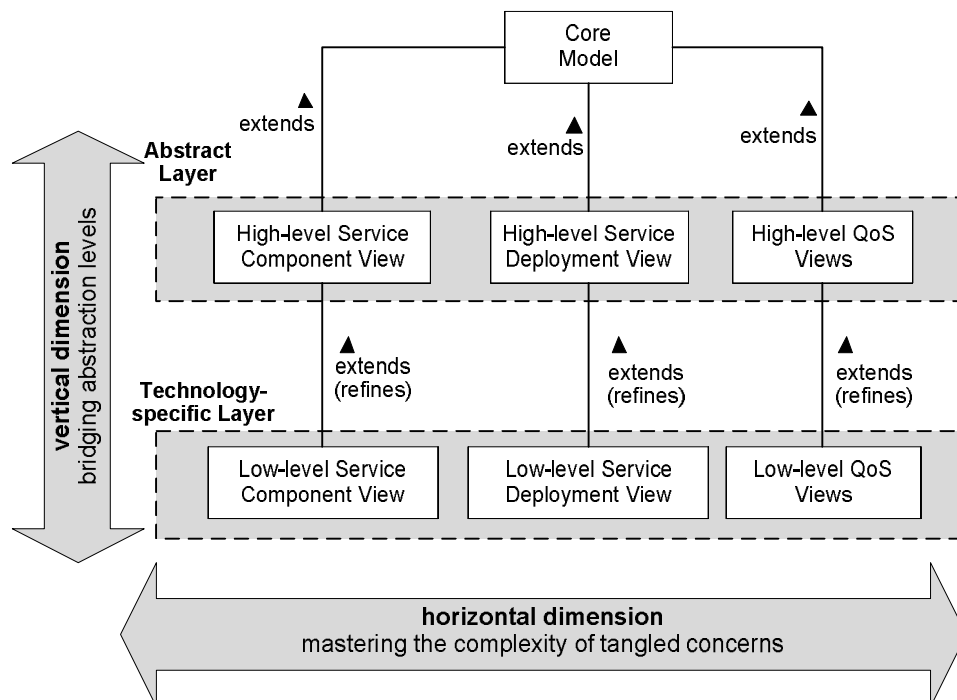


Figure 4 Overview of the view-based design time and runtime architecture

The notion of architectural view has been widely used for efficiently managing the complexity of software system models [RW2005, THZ+09, CBB+2010]. The complexity of modern, especially large-scale software systems make them difficult to grasp all at once. Instead, we restrict our attention at any one moment to one (or a small number) of the software system's structures, which we represent as views. A view is a representation of a set of system elements and the relations associated with them [RW2005, CBB+2010].

We propose a view-based architecture to support modelling various aspects of a virtual service platform. Leveraging the view-based architecture, stakeholders will be able to work with view models that are more appropriate to their expertise. For instance, software architects might leverage high-level abstractions to communicate with the business analysts or customers while developers merely work with low-level, technology-specific descriptions.

The Model-driven Software Development (MDS) paradigm [SV2006] provides a potential solution to this problem by separating the platform-independent and platform-specific models. A platform-independent model is a model of a software system that does not depend on the specific technologies or platforms used to implement it while a platform-specific model links to particular technologies or platforms. Leveraging this advantage of the MDS paradigm, we devise a model-driven stack that has two basic layers: *abstract* and *technology-specific* (see Figure 4). The abstract layer comprises the view models abstracted from the technical details whilst the technology-specific layer contains the views that embody concrete information of technologies or platforms.

A low-level, technology-specific view model can be directly derived from the Core model or based on an existing abstract one. By refining an abstract layer down to a technology-specific layer, this view-based architecture can help bridging the abstraction levels along the vertical dimension, i.e., the dimension of abstraction [THZ+09].

Based on the specifications of view model, stakeholders can create different types of views for describing the services and service collaborations that constitute a particular VSP. A new concern can be integrated into the view-based architecture by defining a corresponding *New-Concern-View* model that extends the basic concepts of the Core model and defines additional concepts of that concern. By adding new view models for additional concerns, we can extend the view-based approach along the horizontal dimension to deal with the complexity caused by the various tangled concerns of a virtual service platform.

The Generation Tools, which implement model-to-code transformation techniques [SV2006], can be used to generate code for VSP such as service descriptions and/or implementations, deployment configurations, monitoring directives, etc. out of the created views. The resulting code and configurations, which may be augmented with hand-written code, can be deployed in an appropriate hosting platform for the aforementioned VSP.

In the subsequent sections, we will describe the specifications of the aforementioned view models as well as the view refinement and code generation techniques in detail.

2.4 Core Model

Aiming at the openness, extensibility, and better integration of the view-based architecture, we devised a Core model as a basis for creating other view models. Figure 5 shows a representation of the Core model using the UML Class Diagram notations [OMG-UML]. The Core model provides a basic set of conceptual elements: *NamedElement*, *Annotation*, *AnnotatedElement*, *View*, and *ViewElement*. These are abstract classes that must be extended further in the extended view models.

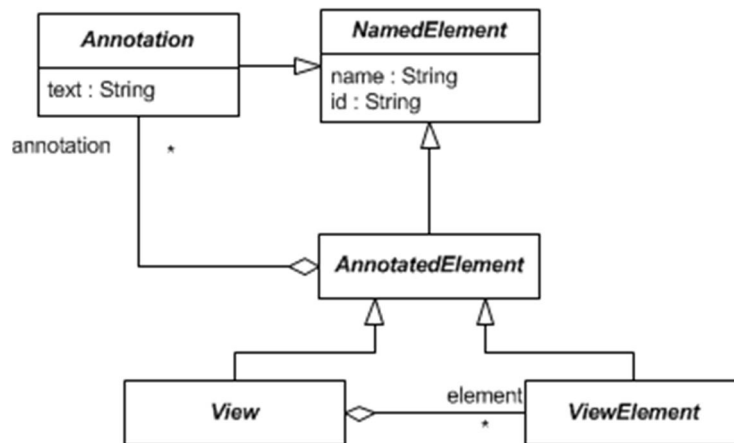


Figure 5 The Core model

At the heart of the Core model is the *View* class that represents the concept of architectural view. A certain *View* might comprise several *ViewElement*. Each specific view element has to concretize the *View* class to represent one particular perspective of the virtual service platform. In other words, view models representing various concerns of a VSP are mostly defined by extending concepts of the Core model. As such, the view models are independent of each other, and the Core model becomes the place where the relationships among the view models are maintained. Hence, the relationships between concepts of the Core model are necessary for extending view models, managing dependencies between views, and generating code.

2.5 Service Component View

Nowadays, component-and-connector (C&C) models have been extensively used in both academia and industry for describing software system architectures [HC2001, Szyperski2002, LW2007, TMD2009]. In a component and connector model, the system is viewed as a collection of entities called *software components*. While executing, a component might need to interact with others. Therefore, a *connector* is used to represent means for the interaction between two components. Examples of connectors are pipes and sockets. Shared data can also act as a connector. If the components use some middleware to communicate and coordinate, then the middleware is a connector [HC2001, Szyperski2002, LW2007, TMD2009].

To provide abstract notions of elements of VSPs, we devise a so-called Service Component View, based on traditional C&C models, at the abstract layer of the view-based architecture in order to formulate VSP architectures. The primary elements of this view are *components* and *connectors* (see Figure 6).

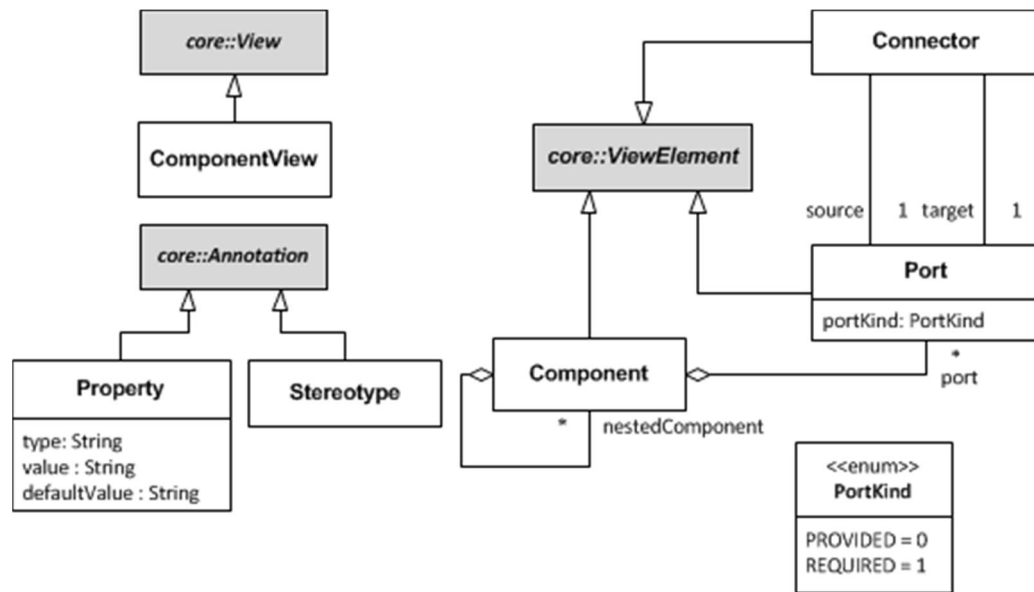


Figure 6 The high-level Service Component View model

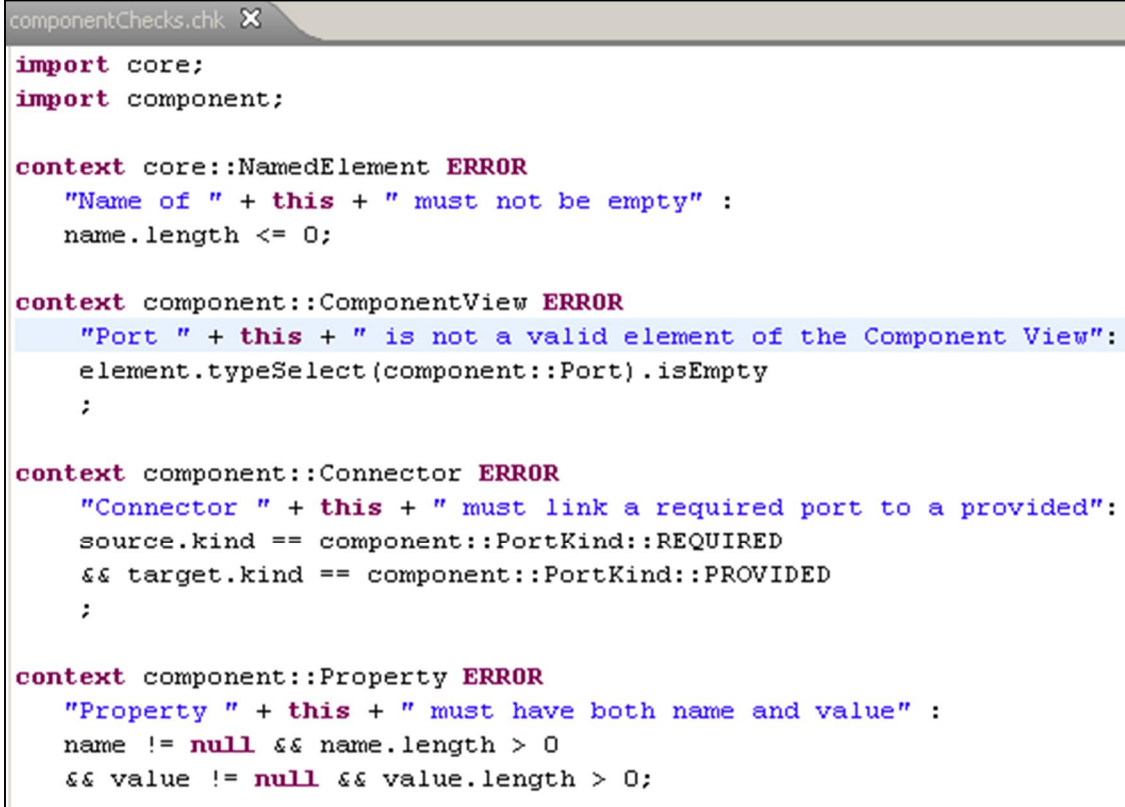
We present a (semi-)formal representation of the Service Component View in Figure 6 using the UML Class Diagram notations. A *component* is the corresponding abstract notion of a *service* which is provided either by a certain service platform or by the virtual service platform. A *connector* represents the interaction between two corresponding components [HC2001, Szyperki2002, TMD2009]. A *component* may contain a number of sub-components. The relationship between a component and its children is represented by the aggregation "*nestedComponent*".

Access to the functionality encapsulated in a certain component is defined through its interfaces, namely, *Ports*. A port of a component can be considered as an abstraction of the corresponding service's interface. There are two kinds of ports. Provided ports represents the functionality that a component exposes to the others whilst required ports are functionality that the component asks for.

In addition, *Properties* and *Stereotypes* can be used in the Service Component View in order to augment the semantics of its constituent elements. For instance, one can assign a certain component with stereotypes "*Web service*" or "*RESTful service*" to indicate that the component is actually a Web service or RESTful service, respectively. Similarly, one can also assign particular properties to a component, connector, or port. These properties can be used for rigorously analysing and reasoning about system architectures.

For validation of Service Component Views at design time, we devise OCL-based rules that are implemented using the Check language of the EMF Xpand/Xtend technologies (see Figure 7) [M2T]. Examples of basic validation rules are the *name* attribute of a NamedElement must exist, ports are not direct children of a Service

Component View, a connector must link a required port to a provided one, a property must have at least name and value attributes, and so on.



```

componentChecks.chk X
import core;
import component;

context core::NamedElement ERROR
  "Name of " + this + " must not be empty" :
    name.length <= 0;

context component::ComponentView ERROR
  "Port " + this + " is not a valid element of the Component View":
    element.typeSelect(component::Port).isEmpty
  ;

context component::Connector ERROR
  "Connector " + this + " must link a required port to a provided":
    source.kind == component::PortKind::REQUIRED
    && target.kind == component::PortKind::PROVIDED
  ;

context component::Property ERROR
  "Property " + this + " must have both name and value" :
    name != null && name.length > 0
    && value != null && value.length > 0;

```

Figure 7 Essential OCL-based rules for validating a Service Component View

In Figure 8 we depict a graphical proof-of-concept representation of the Service Component View implemented using Eclipse Modelling Technologies such as EMF [EMF], GEF [GEF], and GMF [GMF]. This is a Service Component View describing a fictional retailer system that uses services provided by other service platforms such as an enterprise resource planning (ERP) platform, a warehouse management system (WMS) platform, and a delivery service platform. Services provided by each platform are abstracted by components, respectively. The interactions between these components are represented by the connectors linking the components' ports. In this way, software architects can leverage Service Component Views to as means for sketching out the basic functionality of software systems (and in particular, virtual service platforms) as well as efficiently communicating with both non-technical and technical stakeholders.

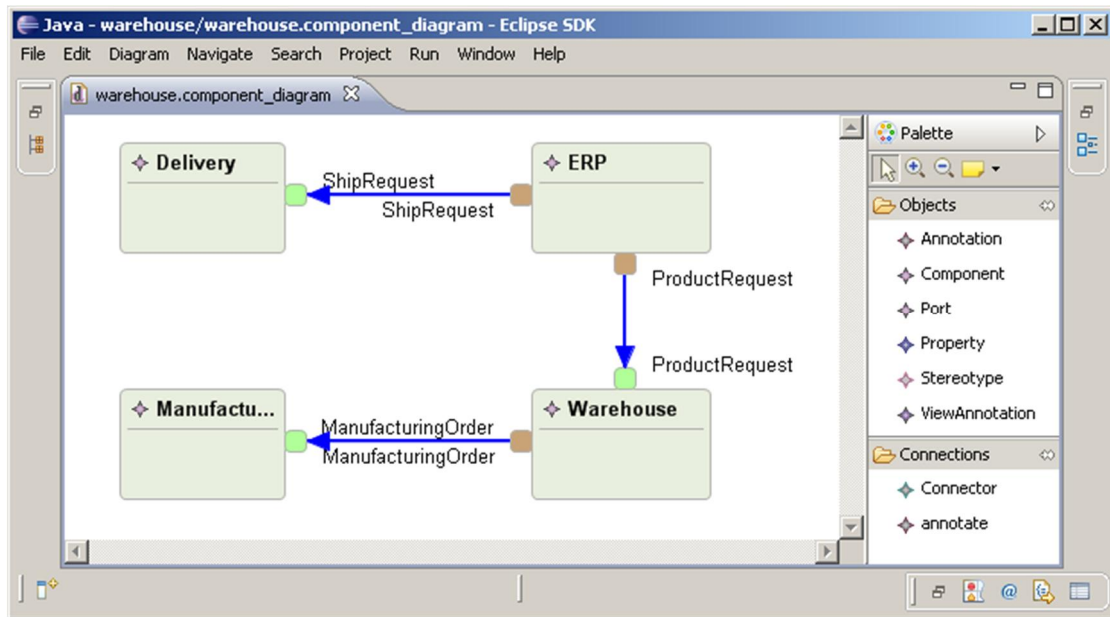


Figure 8 An example of the Service Component View

2.6 The Service Deployment View

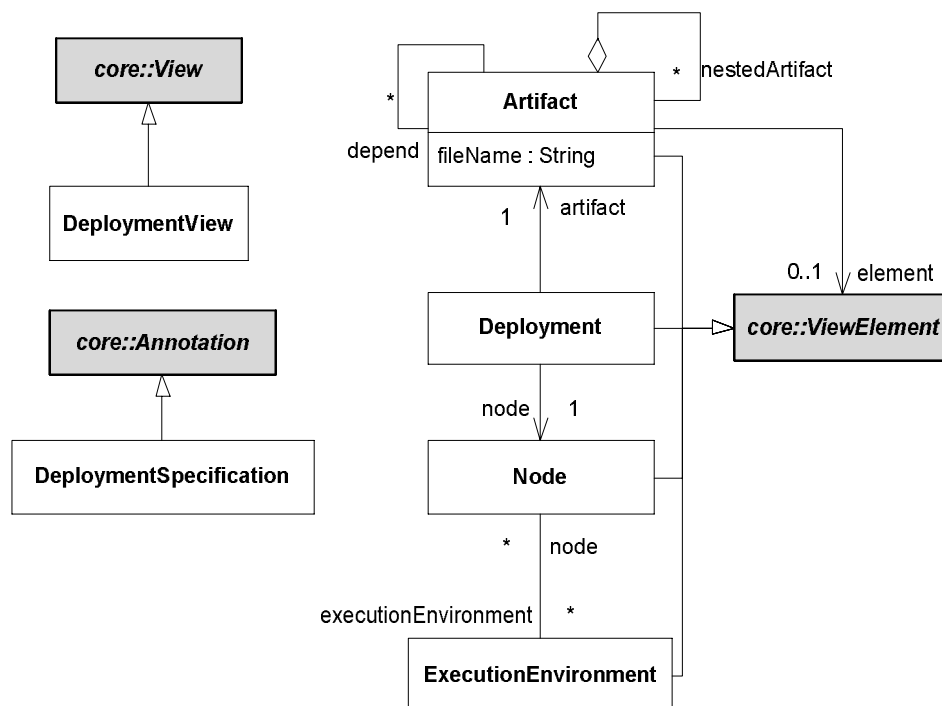


Figure 9 The Service Deployment View

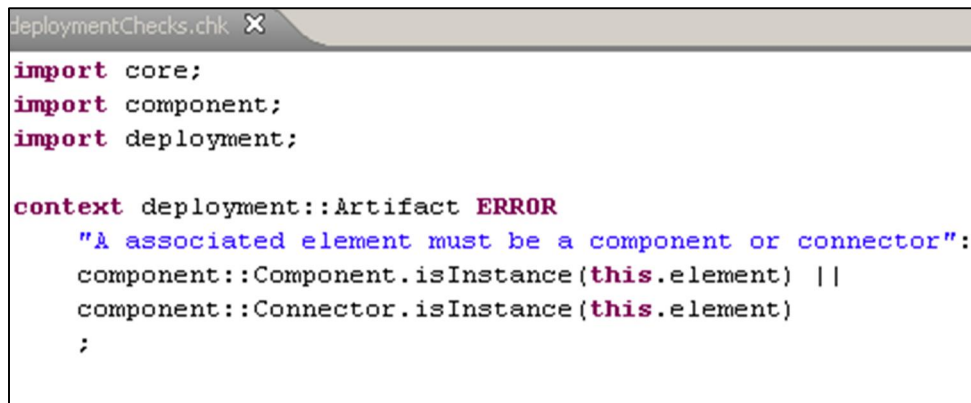
The Service Deployment View provides another perspective of a virtual service platform (see Figure 9). It specifies *where*, and probably *how*, artefacts of a virtual service platform are going to be deployed at runtime [OMG-UML] according to the abstractions provided in the Service Component View. We provide abstract concepts for representing the deployment perspective based on the UML 2 Deployment Diagram [OMG-UML] which is widely used in both academia and industry for the aforementioned purpose. An UML 2 Deployment Diagram can be seen as a concrete refinement of this Service Deployment View. As a result, the integration with,

exchange, or reuse existing UML 2 Deployment models and tooling can be possibly conducted with reasonable efforts.

Similar to the Service Component View, the Service Deployment View is a specialisation of the abstract View defined in the Core model. The elements constituting the Service Deployment View are sub-classes of the ViewElement accordingly. A Service Deployment View can comprise a number of *Deployment* elements each of which associates an *Artifact* and a *Node*.

An Artifact comprises an association to an architectural element such as a component (e.g., a Web service, an OSGi service, etc.) or even a complex connector (e.g., a shared file, a tuple space, etc.). An Artifact has an optional attribute *fileName* that indicates the path of the file containing that Artifact.

A Node represents an appropriate hosting element in which the corresponding Artifact can be deployed and executed. Examples of a typical Node include a Web server, an application server, a business process engine, an enterprise service bus, and so on.



```
deploymentChecks.chk X
import core;
import component;
import deployment;

context deployment::Artifact ERROR
    "A associated element must be a component or connector":
    component::Component.isInstance(this.element) ||
    component::Connector.isInstance(this.element)
    ;
```

Figure 10 Essential OCL-based rules for validating the Service Deployment View

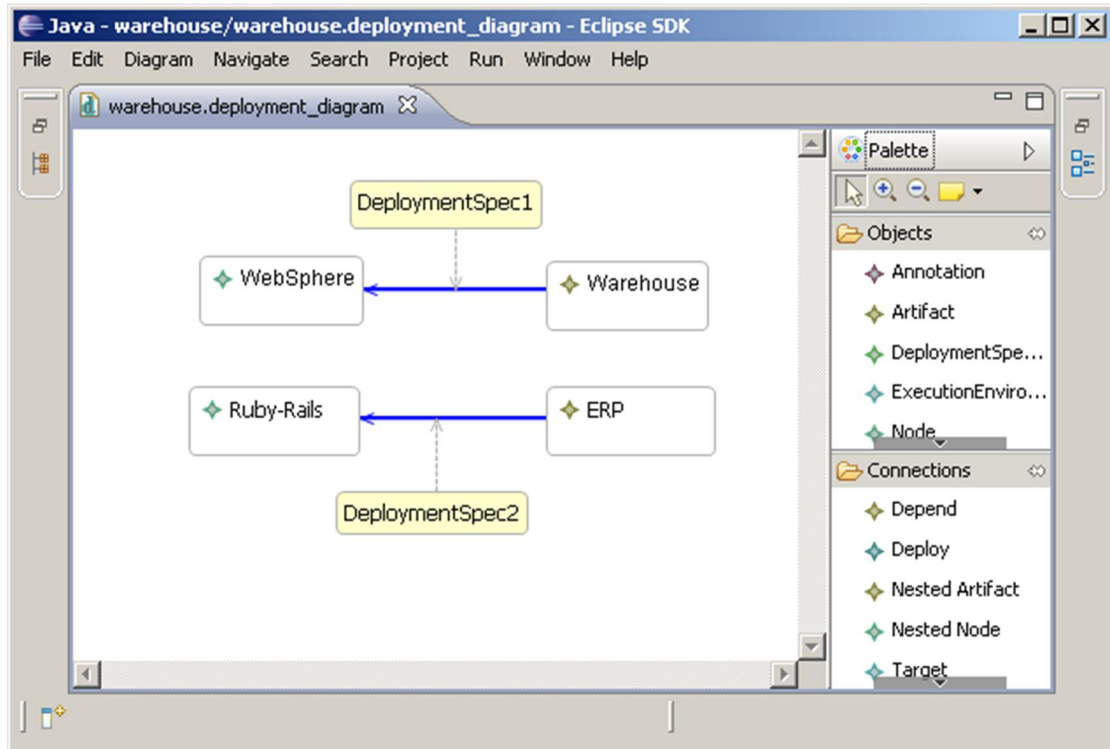


Figure 11 An example of the Service Deployment View

We also define rules that are essential for validating a Service Deployment View (see Figure 10), for instance, an Artifact must refer to an element of type either component or connector. In Figure 11, we present a proof-of-concept implementation of the Service Deployment View in terms of a graphical editor using the same technologies as those used to implement the Service Component View. The Service Deployment View shall be extended and used in WP4, in particular, the Deployment Manager, for deploying runtime artefacts of virtual service platforms. We will discuss further about the integration between WP3 and WP4 on service and virtual service platform deployment in Section 5.6.

3 Runtime Realization and Code Generation

3.1 Extension mechanisms

These abstract views aim at capturing high-level domain-related concepts, and therefore, they are in the first place potentially useful for enhancing the communication with non-technical stakeholders. Nonetheless, the developers often need more information, especially platform- and technology-specific descriptions. According to the specific requirements on the granularity of the views, we can refine these views toward more concrete, technology-specific views using extension mechanisms [THZ+09].

A view refinement is performed by, firstly, choosing adequate extension points, and consequently, applying extension methods to create the resulting view. An extension point of a certain view is a view's element that is enhanced in another view by adding additional features (e.g., new element attributes, or new relationships with

other elements) to form a new element in the corresponding view. Extension methods are modelling relationships such as generalisation, extend, etc., that we can use to establish and maintain the relationships between an existing view and its extension. In the subsequent sections, we introduce a refinement of the high-level Service Component View presented in Section 2.5 for Service Component Architecture (SCA) technology.

3.2 Low-level, technology specific views

3.2.1 Introduction to Service Component Architecture (SCA)

Service Component Architecture (SCA) is a set of specifications for building software systems using a service-oriented architecture. Leveraging concepts derived from C&C models, SCA enables the development of service-based systems based on components that offer their capabilities through service-oriented interfaces and/or consume functions offered by other components also through service-oriented interfaces [OSOA].

One of SCA's advantages is to enhance the decoupling of service implementation and of service assembly from the details of infrastructure capabilities and from the details of the access methods used to invoke services. On the other hand, SCA aims at supporting service implementations written using any one of many programming languages including conventional object-oriented and procedural languages (e.g., Java, C++, COBOL), process-centric languages (e.g., BPEL), scripting languages (e.g., JavaScript), declarative languages (e.g., SQL), and so on. Interactions between SCA components include a wide range of remoting binding mechanisms such as Web services, messaging systems, and CORBA IIOP [OSOA].

We exemplify SCA for the technology-specific layer of the view-based architecture. Other runtime technologies, for instance, Open Services Gateway initiative framework (OSGi) [OSGi], are applicable in the view-based architecture as well with reasonable efforts. Figure 12 depicts an example of an SCA system built upon a number of SCA Composites, SCA Components, and SCA Bindings.

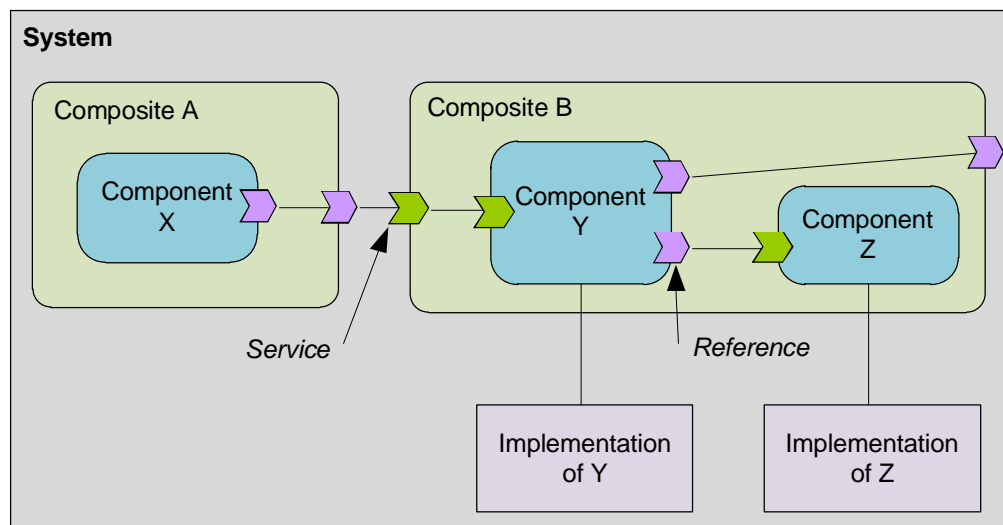


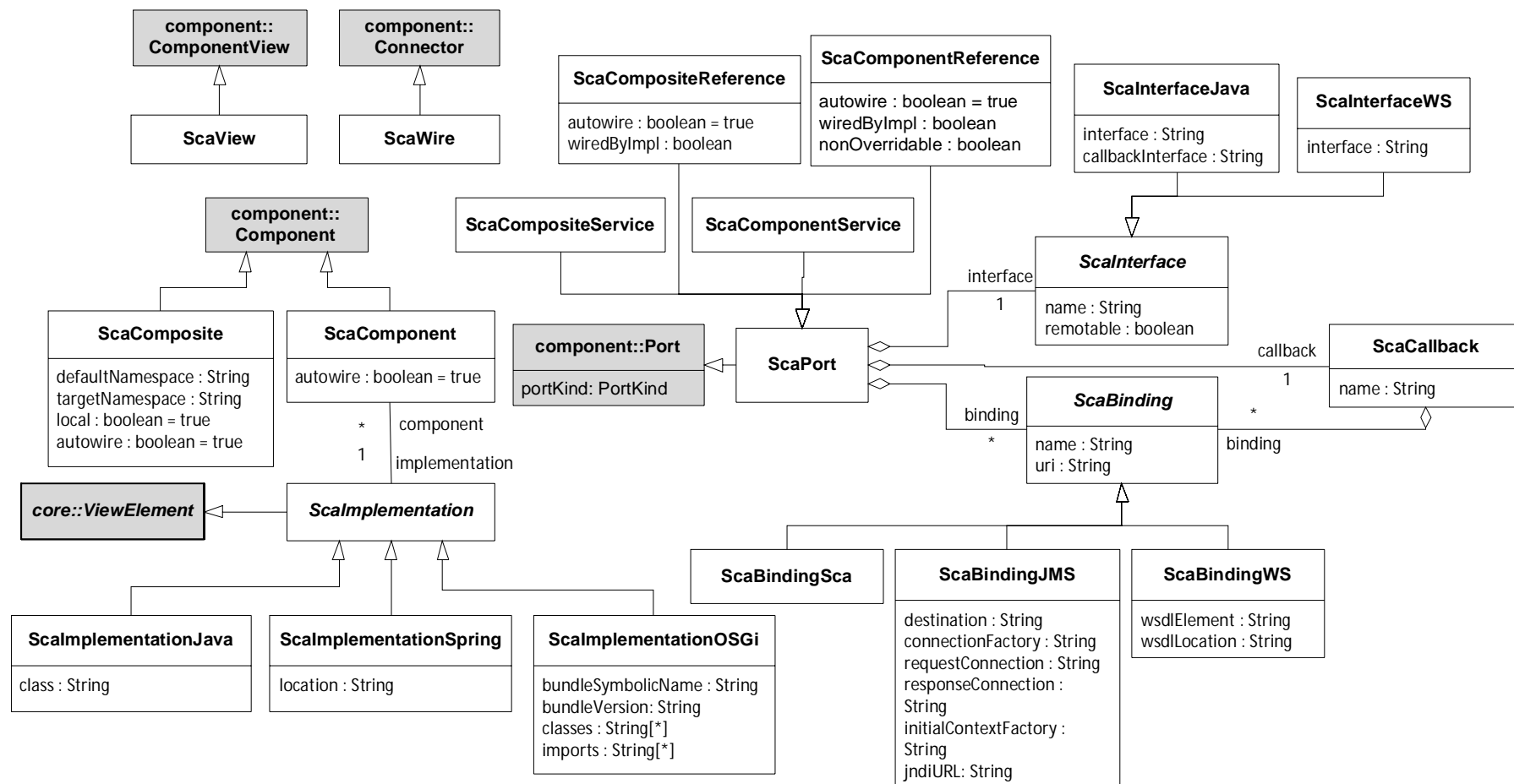
Figure 12 An example of an SCA-based system

3.2.2 The Low-level Service Component View for SCA

In Figure 13, we present the low-level Service Component View for SCA technology. This is an extension of the high-level Service Component View. Thus, a number of concepts extend the corresponding concepts of the high-level counterpart. For instance, the SCA Composite and SCA Component are sub-classes of the Component. References and Services of SCA Composite and SCA Components are sub-classes of the Port.

We also add new elements and attributes to represent the specificity of the SCA technology such as the SCA Implementation, SCA Interface, SCA Binding, SCA Callback, and so on.

Figure 14 illustrates an example of the low-level Service Component View containing elements refined from the high-level counterpart for the same retailer system. Note that the components have been refined accordingly to SCA Composites or SCA Components, respectively, and there is a root SCA Composite embracing these elements.



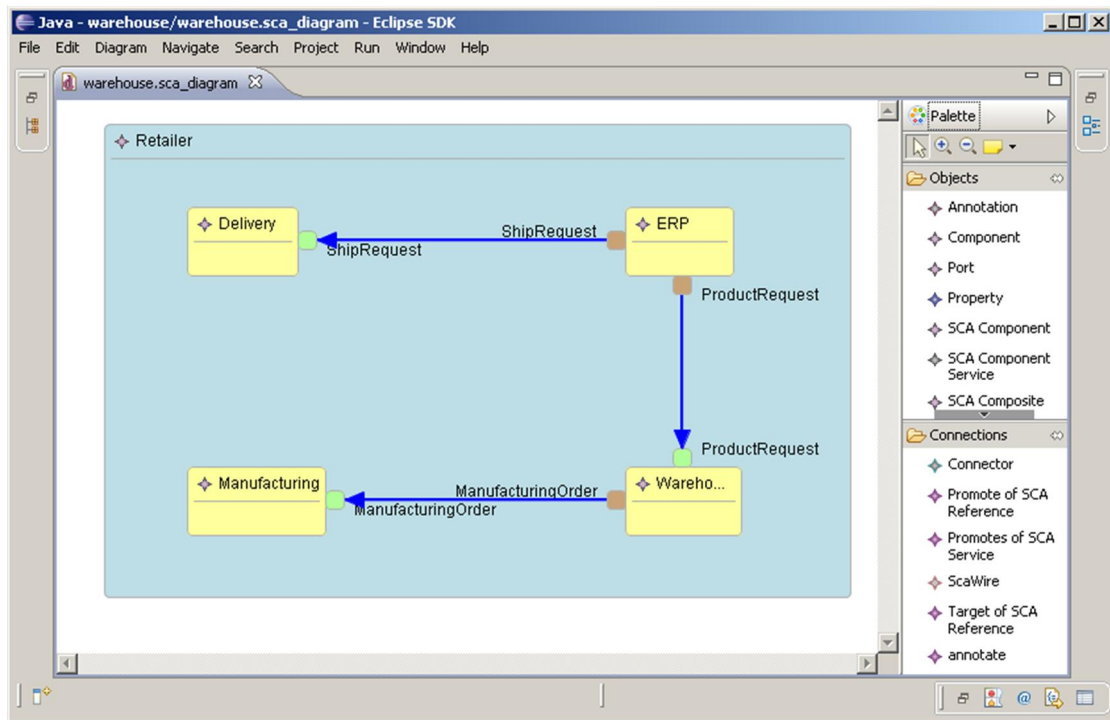


Figure 14 An example of the low-level Service Component View for SCA

3.2.3 Refining high-level to low-level Service Component View

Creating the low-level Service Component View from scratch is a time-consuming task. Given an existing high-level Service Component View, we propose model-to-model techniques to help stakeholders to quickly achieve a low-level Service Component View. Obviously, the resulting low-level Service Component View will miss technical details that are not possible to infer from the abstract, high-level Service Component View. Nevertheless, it can server as a starting point that the software architects and/or developers can add further technical details that are specific to SCA.

Note that SCA allows only one nesting level. That is, an SCA Composite can contain only SCA Component and an SCA Component is not allowed to have sub-component. In addition, an SCA Component is not allowed to be a standalone element, i.e., it must be contained inside an SCA Composite.

As such, our model-to-model transformation rules will focus on two essential strategies: (1) transforming non-nested high-level components to low-level counterparts and (2) transforming one-level nested high-level components to low-level ones. Figure 15 illustrates the former strategy and Figure 16 depicts the later.

In summary, a non-nested component shall be transformed into an SCA Component that is wrapped inside an SCA Composite. A component contains a sub-component shall be mapped into an SCA Composite whilst its child shall be transformed into the SCA Component contained inside that SCA Composite. In both cases, components' ports will be transformed and wired accordingly. We develop a proof-of-concept implementation of the aforementioned view refinement using the Eclipse Xtend

technology [M2T]. More details of the Xtend transformation rules will be shown in the Appendix B.

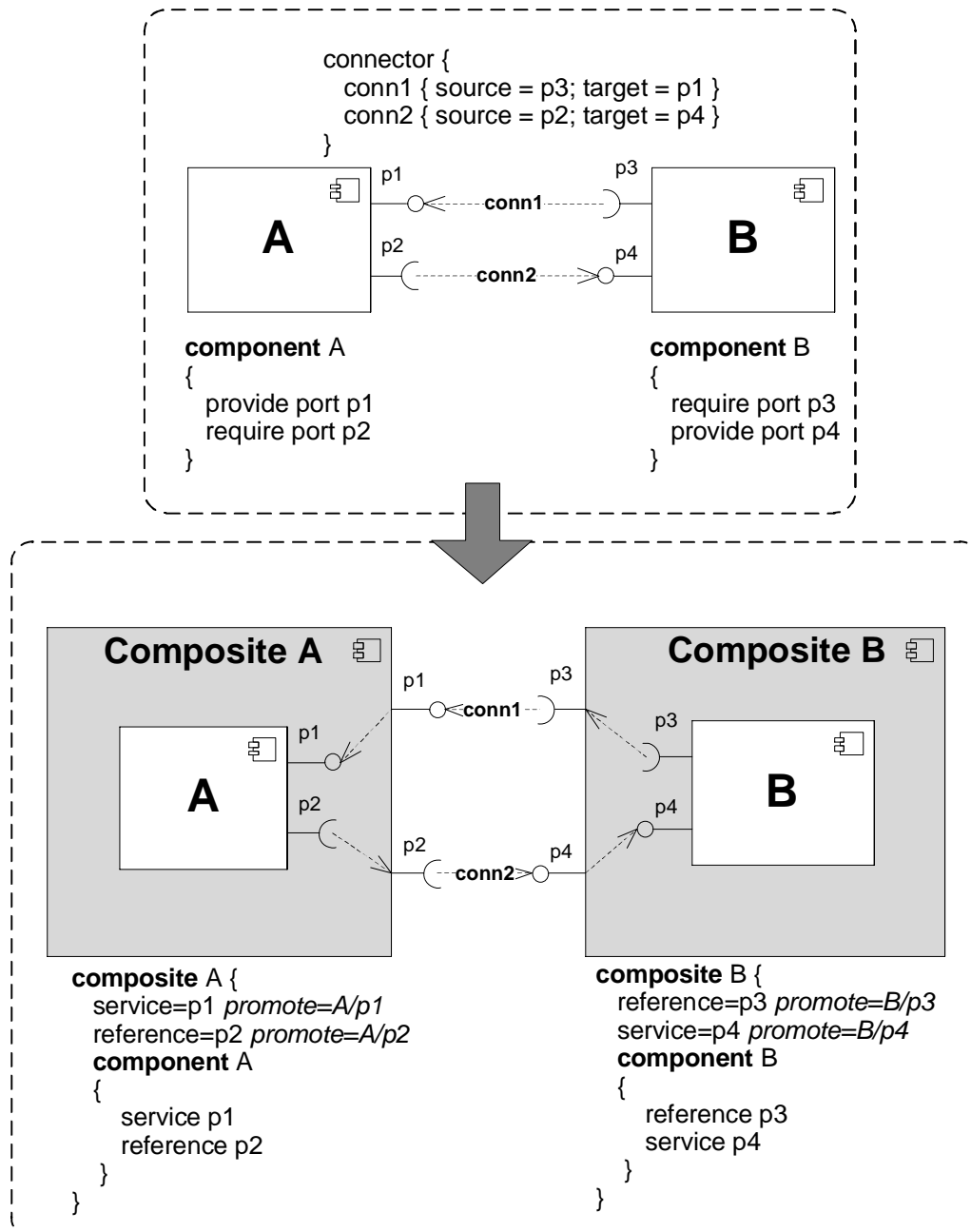


Figure 15 Refining non-nested high-level components to low-level components

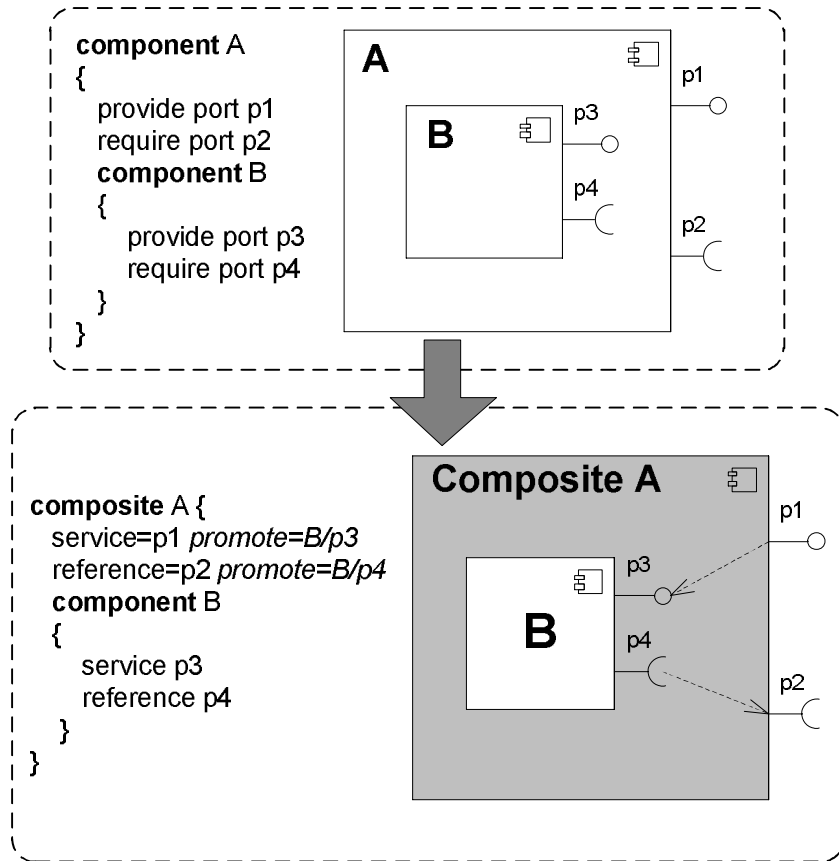


Figure 16 Refining nested high-level components to low-level components

3.3 Runtime View

The Runtime View aims at enabling stakeholders to describe the concepts that are necessary at runtime such as QoS and monitoring properties, adaptation strategies, etc. These concepts can be used, on the one hand, to capture the contractual agreement between a service provider and a service consumer, and, on the other hand, to specify corresponding runtime *Measurements* and/or *Actions* that can be taken when a certain violation occurs. The contractual agreement is so-called a service level agreement (SLA) [SLA]. Service level objectives (SLOs) are key elements of an SLA [FEM+2007]. SLOs are specific measurable characteristics of the SLA such as availability, throughput, response time, etc., [FEM+2007]. An example of a high-level specification of the QoS property of an SLO is: “*The availability of the service X must be greater than or equal to 99%*”.

Figure 17 illustrates the concepts of the Core model being extended in the Runtime View. Each measurable element of the type *ViewElement* involved in an SLO will be annotated with a QoS measurement. To precisely constraint the annotation of QoS properties, we can define similar OCL-based rules as those for the Service Component View and Service Deployment View models. Furthermore, elements of the type *ViewElement* can be annotated with actions that are to be performed when the QoS measurements violate an SLO. The actions to be performed aim to ease the adherence to SLOs, either passively by notifying the operator of the violation (*NotificationAction*) or actively by performing corrective changes to the running system, to autonomically mitigate and/or prevent further violations from occurring

(*AdaptationAction*). In the figure, three example adaptation actions are depicted: *ChangeServiceBinding* means that one service client in the VSP is now rebound to a different service provider. *SwitchVariant* means that an alternative variant of a component is used, e.g., a component produced by selecting different features using the variability model researched in WP2. Finally, *RegenerateAndRedeploy* captures a very intrusive type of adaption, where configurations in the view-based model are adapted, and the resulting code is generated anew from the updated models. Nevertheless, we implement the Runtime View using Eclipse Xtext technology [Xtext]. Xtext is a powerful modelling framework aiming at supporting the development of domain-specific languages (DSL) providing appropriate textual syntaxes for domain experts, technical experts, and/or non-technical stakeholders. Xtext can be well integrated with other Eclipse modelling technologies such as EMF [EMF], Xpand and Xtend [M2T] that we leveraged for developing the proof-of-concept implementation so far. Furthermore, Xtext also supports nice error and warning messages that are seamlessly integrated with the Eclipse IDE.

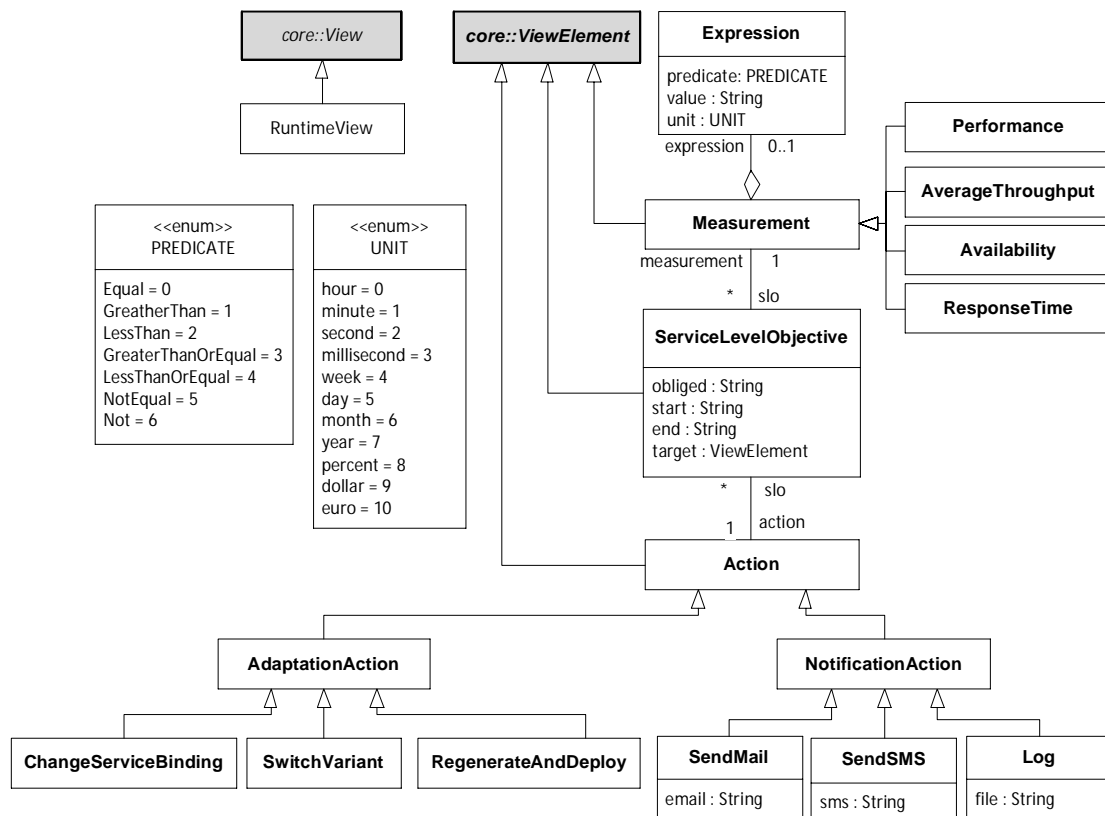


Figure 17 The Runtime View model

The formal grammar of the language is presented in detail in the Appendix A. In Figure 18 we illustrate an excerpt of the Runtime View in the textual syntax developed in Xtext.

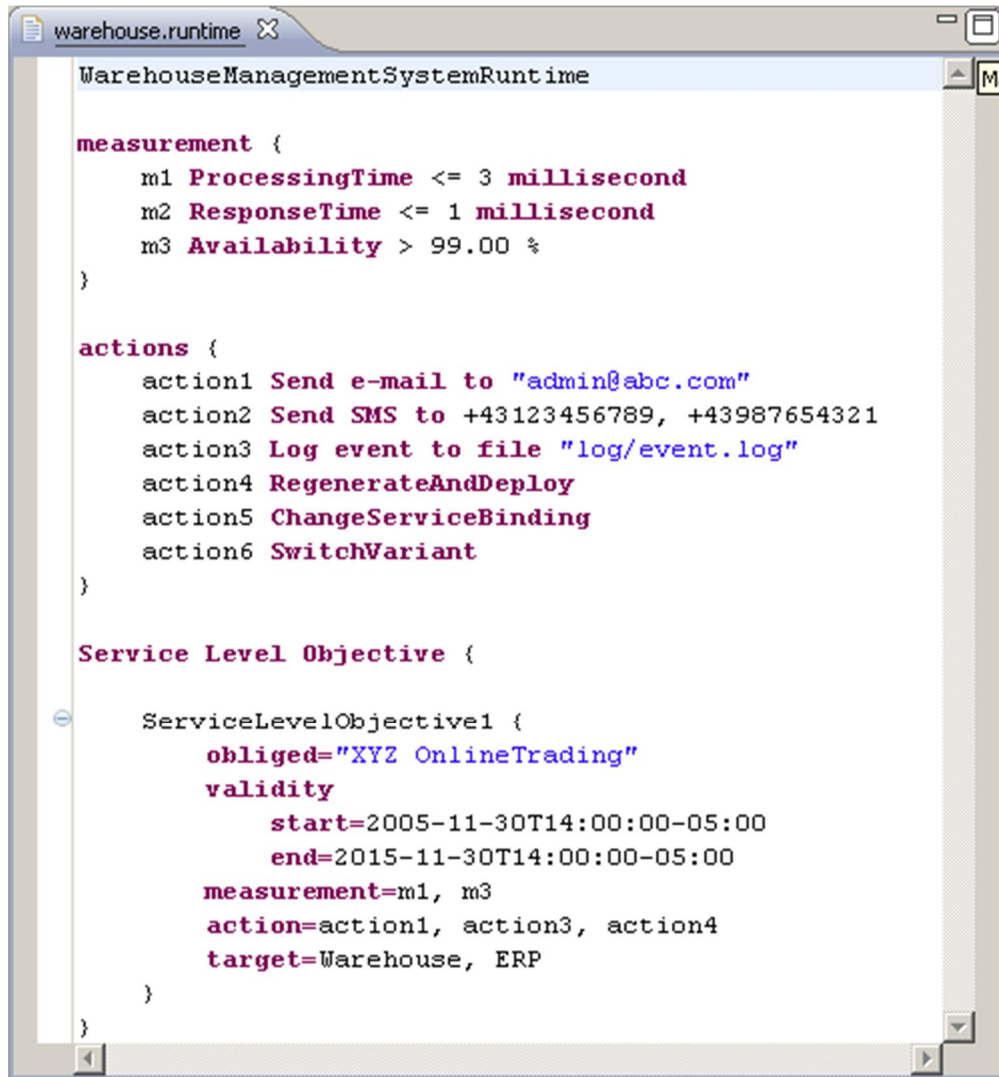


Figure 18 An example of Runtime View in textual syntax

The concepts of the Runtime View can be used for enhancing the communications among multiple stakeholders such as domain experts, software architects, developers, and probably users. As a result, these are rather at high-level of abstraction. Therefore, the Runtime View shall be adequately mapped into low-level representations or even into implementations using specific runtime technologies by using, for instance, methods and techniques presented in [OZD2010]. Extensions of the Runtime View designed in this deliverable shall aim at supporting runtime monitoring and adaptation methods and techniques in WP4. We will discuss further in the integration between WP3 and WP4 in Section 5.5.

3.4 Code Generation

There are two basic types of model transformations: model-to-model and model-to-code [SV2006]. A model-to-model (M2M) transformation maps a model to another model. Model-to-code (M2C), so-called code generation, often produces schematic recurring, and maybe executable, code, that makes up the software products from the models. In both types of transformation, the transformation rules are often defined, firstly, based on the source model. In addition, the transformation rules in

M2M require the specification of the target model while the transformation rules in M2C may need specific platform-definition models.

In the view-based architecture, model-to-code transformation is used to generate VSP code out of one or many input views. In the literature, there are different code generation techniques such as template-based transformation, inline generation, or code weaving have been proposed [SV2006]. In our proof-of-concept implementation, we exemplify the *template-based technique* realised using the Xpand language [M2T] to implement the code generations.

In the following listing, we show an excerpt of the Xpand rules for generating SCA code out of the low-level Service Component View for SCA.

```
«DEFINE COMPOSITE(List[sca::ScaComposite] composites) FOR List»
  «FOREACH composites AS composite»
    «IF composite != null && composite.name != null
      && composite.name.length > 0»
      «FILE composite.name + ".composite"»
      <?xml version="1.0" encoding="UTF-8" standalone="no"?>
      <sca:composite name="«composite.name»"
        xmlns:wsdl="http://www.w3.org/2004/08/wsdl-instance"
        xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
        xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.0"
        «IF composite.targetNamespace != null
          && composite.targetNamespace.length > 0»
          targetNamespace="«composite.targetNamespace»"
        «ELSE»
          targetNamespace="http://«composite.name.toLowerCase()»"
        «ENDIF»»
        «EXPAND COMPONENT(composite.nestedComponent)»
      </sca:composite>
      «ENDFILE»
    «ENDIF»
  «ENDFOREACH»
«ENDDEFINE»

«DEFINE COMPONENT(List[sca::ScaComponent] components) FOR List»
  «LET getScaDSL() AS ScaDSL»
  «FOREACH components AS component»
    «IF component != null && component.name != null
      && component.name.length > 0»
      <sca:component name="«component.name»">
        «FOREACH implementation.select(i|
          i.component.contains(component.name)) AS impl»
          «EXPAND SCA_IMPLEMENTATION(impl)»
        «ENDFOREACH»
        «FOREACH component.port AS port»
          «EXPAND SCA_PORT(port)»
        «ENDFOREACH»
      </sca:component>
    «ENDIF»
  «ENDFOREACH»
«ENDLET»
«ENDDEFINE»

...
```

And an excerpt of the SCA configuration generated using the aforementioned template rules is following.

```
<?xml version="1.0" encoding="UTF-8"?>
<sca:composite
  xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.0"
  xmlns:wsdl="http://www.w3.org/2004/08/wsdl-instance"
  name="Retailer"
  targetNamespace="http://retailer">
  ...
  <sca:component name="Manufacturing">
    <sca:implementation.java
      class="wms.manufacturing.impl.ManufacturingImpl"/>
    <sca:service name="ManufacturingOrder">
      <sca:binding.ws/>
      <sca:binding.sca/>
    </sca:service>
  </sca:component>
  <sca:component name="ERP">
    <sca:implementation.java class="wms.erp.impl.ERPImpl"/>
    <sca:reference name="ProductRequest"> </sca:reference>
    <sca:reference name="ShipRequest"> </sca:reference>
  </sca:component>
  <sca:component name="Warehouse">
    <sca:implementation.java
      class="wms.warehouse.impl.WarehouseImpl"/>
    <sca:service name="ProductRequest">
      <sca:binding.ws/>
      <sca:binding.sca/>
    </sca:service>
    <sca:reference name="ManufacturingOrder"> </sca:reference>
  </sca:component>
  <sca:component name="Delivery">
    <sca:implementation.java class="wms.delivery.impl.DeliveryImpl"/>
    <sca:service name="ShipRequest">
      <sca:binding.ws/>
      <sca:binding.sca/>
    </sca:service>
  </sca:component>
</sca:composite>
```

The generated SCA code can be deployed to execute on any implementation of SCA runtime that is compatible to the SCA specification V1.00 such as Apache Tuscany 1.6¹. To ease this task for stakeholders, we also develop transformation rules to generate a simple launcher in Java that can load and deploy the bespoke SCA configuration in Apache Tuscany 1.6.

```
«DEFINE LAUNCHER(List[sca::ScaComposite] composites) FOR List»
  «FOREACH composites AS composite»
    «IF composite.name != null && composite.name.length > 0»
      «FILE "launcher/" + composite.name + "Launcher.java"»
      package launcher;
      import org.apache.tuscany.sca.host.embedded.SCADomain;
      public class «composite.name»Launcher {
```

¹ <http://tuscany.apache.org>

```
        public static void main(String[] args) {
            SCADomain domain =
                SCADomain.newInstance("«composite.name».composite");
            System.out.println("SCA started (press enter to
shutdown)");
            System.in.read();
            domain.close();
            System.out.println("SCA stopped");
        }
    }
    «ENDFILE»
«ENDIF»
«ENDFOREACH»
«ENDDEFINE»
```

The resulting Java code that can be directly executed is as following.

```
package launcher;

import org.apache.tuscany.sca.host.embedded.SCADomain;

public class WarehouseLauncher {
    public static void main(String[] args) {
        SCADomain domain = SCADomain.newInstance("Warehouse.composite");
        System.out.println("SCA started (press enter to shutdown)");
        System.in.read();
        domain.close();
        System.out.println("SCA stopped");
    }
}
```

4 Roles of various stakeholders

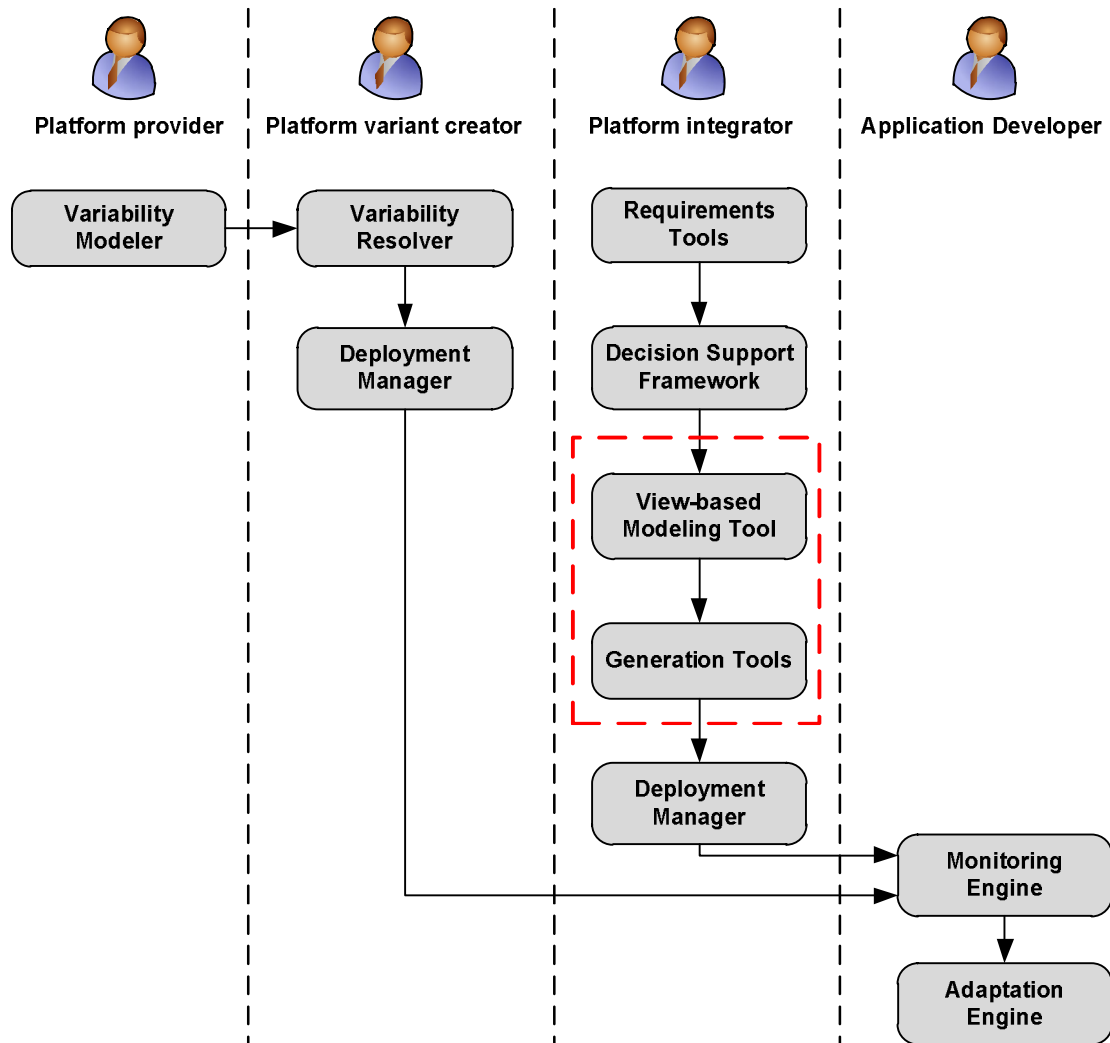


Figure 19 Overview of different roles of stakeholders in INDENICA architecture

The view-based architecture and mechanisms mentioned above are the essential parts shaping the view-based design time and runtime architecture for virtual service platforms in INDENICA. The architecture and accompanying tools aims at supporting *Platform Integrators* in modelling and implementing VSPs as being highlighted in Figure 19.

The inputs of the View-based Modelling Tool are requirements for VSPs that have been interpreted and translated into corresponding architectural decisions and constraints by the *Decision Support Framework*. According to the bespoke requirements, the software architects and developers will start describing functional properties of the VSPs, for instance, by using high-level and low-level Service Component Views (c.f. Section 2.5 and 3.2.2), as well as non-functional properties and runtime monitoring and adaptation strategies, for instance, by using the Runtime View (c.f. Section 3.3). The deployment of VSPs can be specified, for instance, using the Service Deployment View (c.f. Section 2.6) and its extensions.

These deployment models become inputs of the *Deployment Manager* that is responsible for loading and deploying VSPs and services into the hosting platforms.

5 Integration with other INDENICA components

5.1 *Integration of View Models with Goal Models*

Goals provide a well-known and widely used solution for requirements elicitation, but they also provide a flexible and customizable means to address the problem at different levels of abstraction, and with different degrees of precision. The interesting aspect, as for INDENICA, is that the usual goals have been extended with the capability of identifying the requirements for variability and also for adaptation. This means that a single notation/approach can be used to elicit functional requirements, non-functional requirements (often known as qualities of service), variability needs, and also adaptation capabilities. All these concepts can be rendered through natural language, and they would only be useful to the professionals involved in the process, but they can also be stated in more precise and rigorous ways, and thus become the inputs for the automatic derivation of different artefacts in the INDENICA solution:

- Functional requirements and qualities of service will feed the architectural decision process;
- Qualities of service and adaptation goals will be used to identify the dimensions that should be monitored at runtime, how they should be probed, and also the corrective actions in case of problems;
- Variability “annotations” will help identify the actual needs in terms of variability and thus they will support the actual modelling of variability.

This is to say that goals provide a well-scoped view to the view-based approach adopted in INDENICA. They define the first view of the INDENICA solution, and its elements are then mandatory to feed the next steps (views). The complete proposal for the Goal-based solution proposed for eliciting requirements is presented in D1.2, which identifies the main concepts and mechanisms; it also proposes a first version of the meta-model defined to describe these concepts.

The integration between the goal model(s) and the other views is mainly at design time, where the artefacts in the requirements domain are “transformed” into their counterparts in the solution space. Since everything is based on sound meta-models, the transformation (integration) of the different artefacts can be done through automated rules, but the actual degree of automation depends on the completeness and richness of the goal model: the more precise and rigorous it is, the more the whole process can be automated. Even if, there have been already some proposals that promise to transform requirements into running systems automatically, we think that a semi-automatic solution is much more realistic, given the different amounts of information available —and needed— in the two spaces. We can also envisage an incremental approach with a decreasing amount of information that is provided by the professionals, and an increasing amount of automation on the transformation process.

5.2 *Integration of View Models with Architectural Decisions*

Architectural Decisions (ADDs) capture knowledge that may concern a software system as a whole, or one or more components of software architecture. In recent years, software architecture is often considered as a set of principal ADDs rather than the components and connectors constituting a system's design [JB2005, TA2005, LK2007, ZGK+2007]. The idea behind this new perspective is to document not only components and connectors but also the design rationale of the architecture as well as to contribute to the gathering of Architectural Knowledge (AK). All these approaches share the problem of a significant extra effort necessary to record AK. INDENICA will address this problem by integrating AK recording with model-driven views, to easier enable a link between the models and decisions from which they originate – which is one of objectives of task T1.3 of WP1.

In this light, design models (aka architectural views) of a certain virtual service platform (VSP) developed using techniques provided by the view-based framework (see Section 2) shall be influenced by the architectural decisions about that VSP. Furthermore, the ADs about the VSP are, in turn, often derived from its requirements. As a result, one can see the ADDs as one of the bridges between a VSP's requirements and its actual designs and/or implementations. Therefore, the integration between the view-based framework and ADs shall be the linking of the decision models to view-based models in order to connect decisions and the models that created following the decisions.

We aim at bridging ADs and architectural views by introducing models capturing the mapping between them. The main objective is to enable traceability and consistency checking between ADDs and architectural views and/or leverage MDSD to potentially automate the generation of initial instances of architectural views that reflects the design decisions. Besides, the generation of formal constraints that can be used for checking the consistency between the ADs and the architectural views might also be supported in order to assure that ADs and views remain consistent.

5.3 *Integration of View Models with Variability Modelling*

Variability Modelling is used to describe a range of potential customizations of a software system. This approach was initially developed in the context of product line engineering. In INDENICA this is used to describe potential customizations of service platforms. This can be applied both to basic technical services of a service platform (e.g., specific monitoring capabilities), as well as for customizing domain-specific parts of a platform (e.g., yard management might not in all contexts cover the same range of capabilities).

The INDENICA approach to variability modelling uses variability decisions as a basis, as described in D2.1. A variability decision is a placeholder (parameter) that can be set to different values, each representing a different platform instance. Industrial size platforms may have hundreds, if not thousands of these decisions. Of course often constraints among these decisions will exist. Within INDENICA we will in particular provide support for composing the variability in different platforms and for deriving partial instances of platforms. While the former can be used to represent

the variability of virtual platforms that are derived by composing several base platforms, the later can be used to derive domain-specific instances of platforms.

The capability to compose partial variable parts also allows to represent components as in the SCA technology individually and then compose them into a combined variable compound. As the composition in INDENICA variability is fully recursive, there is no such restriction as in SCA where only one level of nesting is possible.

The variability-modelling platform in INDENICA will support comprehensive variability management. As part of this it will also support the specification of instances of the generic (parameterized) platforms. It will also be possible at any time to query the corresponding values of the decision of the relevant product line. Thus, one potential approach to mapping variability to the multi-view framework will be to specify variability information for Named elements, e.g., in the form of an additional attribute or annotation²stereotype. The editor would then be able to query the necessary information at runtime to also form instances of a platform view. (Such instances would ever parameterize certain elements or even remove certain elements.)

Instantiation has complex consequences. Variability may lead to certain parts implemented differently or even not at all. This is particular important when the runtime view is generated. A simply way to handle this will be to make the variability explicit in the transformation process described in Section 3.4. The INDENICA variability management environment will also be able to manage this transformation and instrument the code generation process illustrated above.

5.4 Integration of View Models with Enterprise Architecture Management

Experience has shown that large systems or software landscapes cannot be developed or planned without considering the underlying business needs since there was a change regarding the consumption of information over the last two decades. Information becomes more and more important and the timely access to the right information may well decide over raise or decline of companies. The importance of time- and cost efficient processes grows and the enterprise IT landscape is no exception. The classic approach to questions like “How can we know what process are affected by withdrawing an IT system?” is to set up a project that eventually finds the answer to this question. This works well but is rather inefficient: if the same question is raised half a year after the first investigation, no one would feel comfortable in taking the old result again. So another project is started. Why is that? The answer is simple: no one knows the big picture of the enterprise architecture. There are IT architects who may focus on the overall application landscape but miss the connection between business processes and IT applications for handling these processes. This is the point where Enterprise Architecture Management (EAM) comes into play. EAM is the answer to the insight that processes are not just something the business does but that processes *are* the business (see Figure 20). EAM is a holistic and enterprise-centred approach that aims at capturing and

² In UML this could also be realized using stereotypes.

maintaining the overall enterprise landscape including people, processes, businesses and technologies. This blueprint of the enterprise's structure offers transparency and provides the opportunity to perform changes in business processes or the IT landscape while being able to control the corresponding risks. Companies that utilize EAM are then able to manage the complexity of large enterprise IT systems much more efficiently and can answer questions like the one mentioned above without needing to set up dedicated projects for retrieving the answers [DR2011].

The term enterprise architecture (EA) stands for a description of all components of an enterprise and the relationships between them and is part of the enterprise architecture management approach. In this context, the term enterprise may refer to a public or private company as well as to a governmental organization (e.g. the department of defence). It is important to note that the term "architecture" is not limited to the information systems used by an organization but also includes people, business entities, information and their relation. Is it therefore more than a description of the enterprise's IT landscape and aims at the alignment of business processes and IT-tools in order to increase the efficiency of operations.

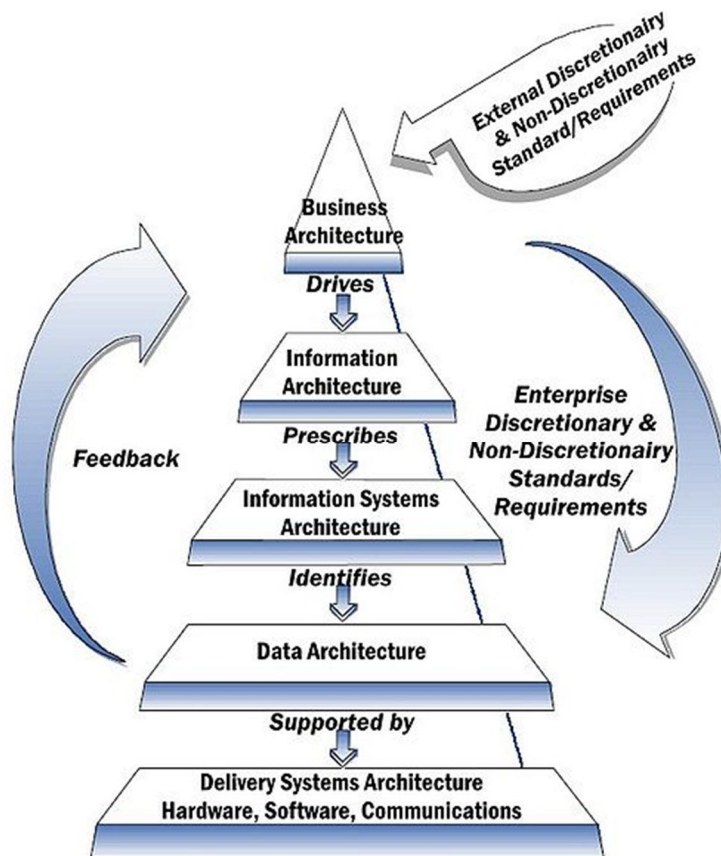


Figure 20 NIST enterprise architecture model
(http://en.wikipedia.org/wiki/File:NIST_Enterprise_Architecture_Model.jpg)

Over the decades many methods and frameworks for enterprise architecture management have been suggested, the following being the most popular ones [SR2007]:

-
- TOGAF³: The Open Group Enterprise Architecture Framework, developed by the Open Group consortium, offers an architecture development method (ADM) and various standards for describing different types of architecture. This is one of the most popular EA frameworks.
 - Zachman Framework: This is one of the first frameworks for describing an enterprise with all its components. In contrast to TOGAF, it does not define any methods or processes for collection of information but rather focuses on the description and organization of the architectural artefacts.
 - FEA: The Federal Enterprise Architecture combines the approaches from Zachman and TOGAF by offering both a comprehensive taxonomy as well as a set of processes for the information collection.

Siemens derived another EAM methodology from the TOGAF framework, as shown in Figure 21. Basically it is divided into four phases:

- Vision and scoping: This phase focuses on the business and IT vision and covers existing guidelines, frameworks and tools. After the scope and the goals for the enterprise architecture assessment have been defined, the EA framework is tailored to meet the goals as efficiently as possible.
- Analysis: in depth knowledge of existing structures, processes, functions, data, applications and their relations is gathered. This information is documented alongside with existing high-level architectures and more technical, low-level architectures.
- Planning: Based on the business vision, business goals, requirements and improvement lists, the target architecture is planned. Core business functions are defined as well as high-level and low-level architectures in form of a service map. Besides these technical artefacts, guidelines like software development processes or architecture principles are suggested.
- Execution. As soon as the target architecture has been defined, a comparison between the as-is and the target architecture is performed. Possible gaps are documented and qualified (e.g. cost and time). A roadmap with the further activities will be compiled which then can be followed in order to implement the necessary measures for reaching the target architecture.

³ <http://www.togaf.info/>

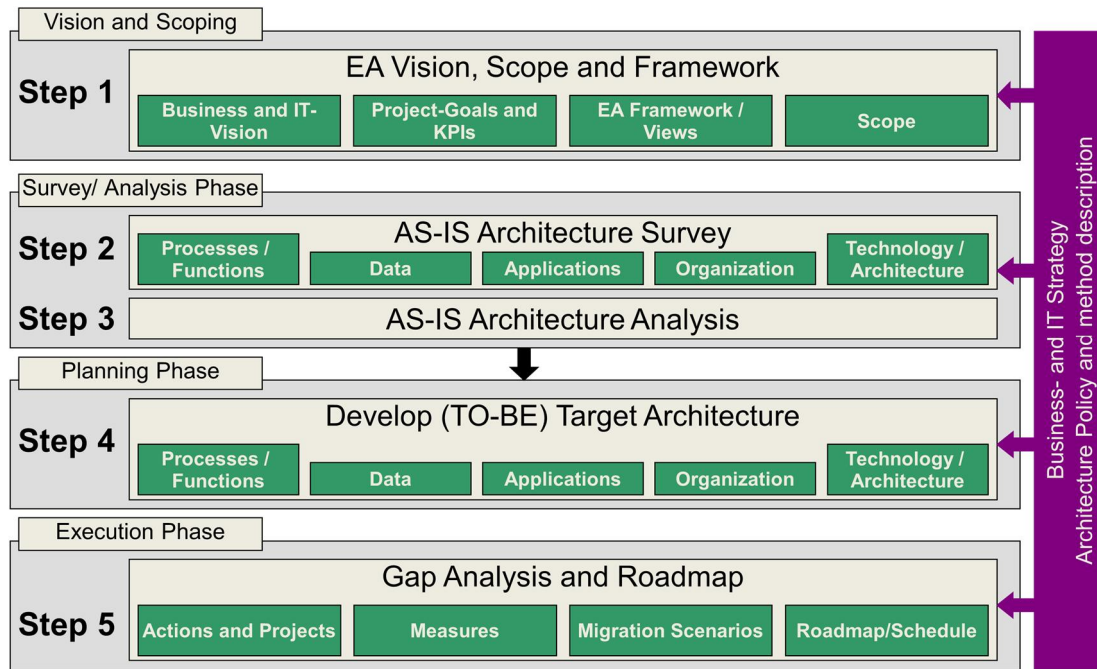


Figure 21: The Siemens Enterprise Architecture Management framework

The architectural views described in Section 2 can be mapped to the EAM development cycle as depicted in Figure 22. Since the views are meant to be used during the definition of the VSP's service landscape they can be considered to be a part of the planning phase. While actual business requirements and business data models are developed and documented using different views and diagrams, the target architecture (logical and functional) and logical data objects like components can be documented using the views described in Section 2.5 and 2.6. The overall platform architecture can be described using the platform architecture view as described in Section 2 leading to the following overlap between the EAM development cycle and the INDENICA Architecture Views:

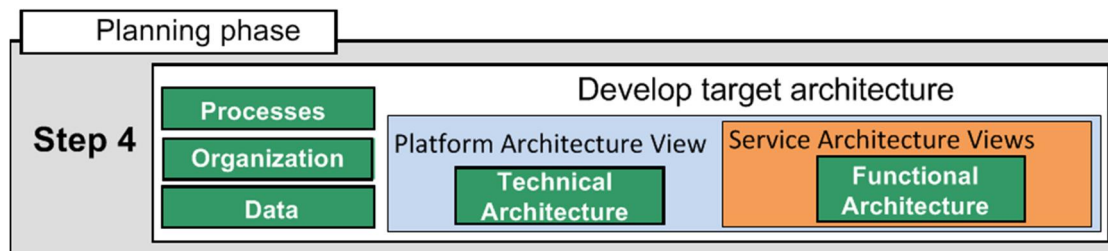


Figure 22: INDENICA architecture views mapped to the planning phase of the EAM development cycle

The platform architecture view covers both the technical architecture (infrastructure and integration components) as well as the (business) functional architecture. The details of the functional architecture are refined by the service architecture views as mentioned in Section 2.5. Processes, organization and data are not covered by any of the views and have to be modelled and managed by other means.

5.5 Integration of Adaptation and Monitoring

In INDENICA, both, VSPs and service platforms, are monitored at runtime, in order to track the health of the system. If performance degradations are detected, adaptation

actions can be executed. The components mainly responsible for those tasks are the Monitoring Engine (sensing the system state in an event-based fashion) and the Adaptation Engine (triggering adaptations, such as reconfiguring service platforms, or regenerating VSP components).

Both the Monitoring Engine and the Adaptation Engine are central parts of the VSP, and are generated from view-based models during deployment. They are modelled as Components in the Service Component View, and their concrete deployment to SCA components is modelled using the Service Deployment View. To this end, the view-based code generation tools described in Section 3.4 are used to generate SCA deployment descriptions, component launch code and SCA composites as necessary. However, most important for the integration of the Monitoring and Adaptation Engine with the view-based modelling framework is the Runtime View.

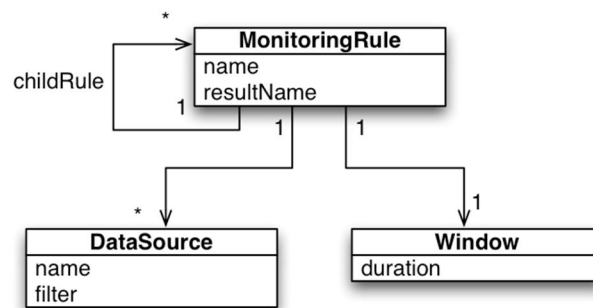


Figure 23: Runtime Model Extension Capturing Monitoring Rules

The INDENICA approach to monitoring is based on top-down refinement of monitoring policies. On the top level (the high-level view in Figure 4), human system operators specify performance goals on a non-technical level, in terms that make sense for domain specialists (e.g., *the VSP needs an availability of 0.9999*). Technical personnel refines these goals semi-automatically into lower-level monitoring objectives, and adds adaptation actions that can be used to improve the system health if the objectives cannot be fulfilled. Finally, in a third refinement step, these (still technology-independent) monitoring objectives and adaptations are mapped to concrete monitorable metrics, aggregation rules for these low-level metrics, and concrete adaptations of the running systems. At runtime, the available adaptations are determined from adaptation actions derived during the top-down refinement process unified with information on unbound variabilities to be resolved at runtime, their dependencies and constraints as specified in the variability model (Section Integration of View Models with Variability Modelling). For instance, a concrete VSP may be using Esper⁴ to implement event-based monitoring. In order to demonstrate how view-based modelling can be used to generate Esper Event Processing Language (EPL) statements, we first need to slightly extend the Runtime View model in Figure 17 (see Figure 23). As additional elements, we introduce monitoring rules (which are composable, i.e., a monitoring rule can be defined as a composition of other rules).

⁴ <http://esper.codehaus.org/>

Every rule has zero or more data sources (rules with zero data sources only make sense in case of composed rules), and exactly one time window. The window defines the duration interval of a measurement.

The process of mapping the monitoring rules defined as instances of this model to the implementation level is illustrated on the basis of a concrete example. The following listing contains an excerpt of the template used to generate the monitoring specification for measuring the availability of a service (i.e., ratio of successful service requests to total service requests). This monitoring specification is represented by an instance of the class `AvailabilityRule`, which serves as the input to the template. The output of the template-based code generation process is a concrete query specification written in EPL, which aggregates the number of logged service invocations (`CountInvocationsRule`) over a certain time window (`rule.window.duration`). The example illustrates the composition of monitoring rules, since `AvailabilityRule` is composed of two instances of `CountInvocationsRule` (one for filtering all failed invocations, and one for counting all invocations) via the association `rule.childRules`. The generation template separates the output definitions of the different types of monitoring rules, and cross-references these definitions where needed (e.g., `outputCountInvocations` is called from within `outputAvailability`), which allows for recursive composition of monitoring rules.

```
«DEFINE main FOR AvailabilityRule»
  «FILE "AvailabilityMonitoringQuery.esper"»
  «EXPAND outputAvailability(this)»
  «ENDFILE»
«ENDDEFINE»

«DEFINE outputCountInvocations(CountInvocationsRule rule) FOR AvailabilityRule»
  select count(invocations)
  «IF rule.resultDescription != null»
    as «rule.resultDescription»
  «ENDIF»
  from «rule.dataSources.get(0).name»
  («rule.dataSources.get(0).filter»)
  «IF rule.window != null»
    .win:time(60*60*«rule.window.duration»)
  «ENDIF»
  as invocations
«ENDDEFINE»

«DEFINE outputAvailability(AvailabilityRule rule) FOR AvailabilityRule»
  select 1 - (
    («EXPAND outputCountInvocations(
      (CountInvocationsRule)rule.childRules.get(0))»)
    /
    («EXPAND outputCountInvocations(
      (CountInvocationsRule)rule.childRules.get(1))»)
  )
  as «rule.resultDescription»
«ENDDEFINE»
```

The resulting EPL looks as follows.

```
select
  1 - (
```

```

(select count(invocations)
from ServiceInvocationFailedEvent(service='ERPSERVICE')
 .win:time(60*60*24.0) as invocations )
/
(select count(invocations)
from ServiceInvocationEvent(service='ERPSERVICE')
 .win:time(60*60*24.0) as invocations ))
as unavail

```

This EPL statement integrates two event streams (the stream of `ServiceUnavailableEvent`, indicating that a service should have been invoked, but was unavailable, and of `ServiceInvocationStartedEvent`, indicating that a service should be invoked) over a time period of one day.

Events emitted by the Monitoring Engine are consumed by the Adaptation Engine. A concrete VSP may be using the Drools Expert⁵ rule engine as basis of the Adaptation Engine. An example DRL (Drools Rule Language) rule looks as follows:

```

rule "When availability too low, increase service redundancy level"
when
    $e : ServiceAvailabilityReportEvent( actualAvailability < 0.9999 )
then
    NotificationActions.notifyOperator($e);
    AdaptationActions.increaseRedundancyLevel();
end

```

5.6 Integration of Deployment

The Deployment Manager which will be developed in WP4 is responsible for deploying the SCA components to the runtime of the Virtual Service Platform. Therefore some deployment descriptors are needed. The SCA deployment descriptors can be divided into three groups. First for the description of the physical structure of the SCA domain, second the logical description of the SCA domain and third SCA metadata. The descriptors for the physical structure specify the SCA domain cloud composed of a set of SCA runtime instances each containing a subset of all SCA composites of the domain including binding details. This information can be derived from the Service Deployment View.

The Deployment Descriptors for the logical structure of the SCA domain is content of the Service Deployment View. One descriptor contains all SCA composites representing the virtual domain (derived from High-Level-View) whereby each SCA composite has a descriptor assembling its SCA components (Low-Level-View). The third and last group contains SCA metadata. At one hand a file describing all SCA contributions of the domain modelled in the Service Deployment View, at the other hand metadata for each SCA contribution that contain information about the usage scope for the contained composites. This information can be derived from the Service Component View. Concrete information about the descriptors can be found in the SCA 1.0 Specification [OSOA] and in the documentation of Apache Tuscany⁶.

⁵ <http://www.jboss.org/drools>

⁶ <http://tuscany.apache.org/>

5.7 Integration of Platform Service

In the second year of INDENICA, we will design and implement the integration with services provided by different service platforms described in WP5. In particular, our architecture and tooling will support modelling and generating necessary service adapters/converters/mediators that can be used to connect the underlying service platforms with the virtual service platform.

A brief description of services provided by the aforementioned platforms is following.

Services provided by WMS Platform (SIE)		
Warehouse Management Service		<p>This service provides basic logic to access the contents of the warehouse.</p> <p>New storage units can be registered prior to the actual storage procedure. Afterwards an appropriate bin location can be searched and reserved. The service also offers a functionality to handle the transport of a storage bin to a reserved bin location. In order to perform a checkout for an order, all storage units that contain a specific article can be searched. Based on this information, a checkout order can be created and processed which ultimately completes the whole checkout procedure.</p> <p>Service consumers may register and unregister for service events like successful storage or arrival of a storage unit at a checkout desk.</p>
Conveyor Control Service		<p>This service provides access to the conveyor belt system that moves storage units within the warehouse building. Based on their unique identifier, storage units can be sent to a bin location, retrieved from a bin location and transported to checkout desks and handover platforms.</p> <p>This service is internally used by the warehouse management service. A usage by external client is not intended.</p>
Services provided by YMS Platform (SAP)		
Yard Management Service		<p>This service provides basic logic to handle common yard management processes.</p> <p>New shipping tasks including their advanced shipping notice can be registered in the system. Arriving truckloads will be scheduled and assigned to loading docks or to a waiting area (dock door scheduling, DDS). Thereby business rules have to be taken into account like docks for oversized goods or docks for rapidly spoiled food, which needs cooling. Also logic</p>

	<p>for rescheduling because of delays is included. Additionally, it provides a basic framework for building a yard management web interface (based on Spring), including a graphical interactive representation of the actual yard.</p> <p>The yard management service will mainly be used by the yard manager for administration and monitoring as well as by the gate guard to register new truckloads and communicate scheduled docks.</p>
Yard Jockey Service	<p>This service allows scheduling of tasks for yard jockeys. These tasks include fetching or relocating trailers on the yard. Additionally, locations of trailers are maintained which allows intelligent scheduling of tasks and optimizes the path of the yard jockey. The trailer position will also be used to select the jockey to which to assign the specific task.</p> <p>By providing generic user interface components, tasks can be created by the yard manager. After creation, the yard jockey will be notified of these new tasks. The jockey can update his status for monitoring purposes.</p>
Mobile Communication Service	<p>This service provides functionality for communicating with mobile devices. To allow fast and effective communication, several persons can be equipped with such devices. This service can be used to distribute notifications and to monitor the state of several yard entities in near real-time. The yard jockey receives notification on new tasks and updates his state whether he is searching for a trailer, carrying a trailer or idling. In case of a delay, the truck driver can send a notification that triggers a rescheduling of the dock occupations. Truck drivers on the yard can receive information about their assigned docks. They will be notified whenever a change occurs. The warehouse staff can update the loading or unloading status of the current trailer easily and receive notifications about new loading tasks.</p> <p>This service also facilitates development of mobile user interface by supplying an application framework for mobile YMS UIs. It thereby provides a development environment for building native apps for mobile devices based on Spring Android.</p>
Location Service	Mobile devices can also be used to communicate

	<p>their position via GPS or similar. This information helps to determine the time till arrival of truck drivers. Additionally, it provides yard jockeys with precise positions about the trailers to be fetched. Because the YMS knows the position of all yard jockeys, it can better assign fetching tasks to the best suited yard jockey.</p>
EDI Service	<p>Electronic data interchange (EDI) allows for standardized information exchange between organizations. This service interfaces the YMS to other organizations by allowing information exchange via EDI. This service will be used by external organizations to transfer advanced shipping notices to the YMS in an electronic way.</p>
Services provided by Remote Management Platform (TARC)	
Call Session management	<p>The service will allow different user to contact other users performing a call or send directed message to one or a group of users. The caller will also automatically provide his/hers current location, and other context information.</p> <p>This service will also include direct messaging which can be easily integrated with emergency services to provide fast one-click emergency application.</p> <p>Callers will be able to share picture or other data needed during the call.</p>
Remote monitoring	<p>The monitoring service will allow responsible staff to check the performance of the whole system in near-real time. Information from various sources will be aggregated and pre-analysed and on-demand reports will be able to be generated in just few seconds. Stored data will also give a chance to perform some data mining in order to produce charts and deeper analysis of trends.</p> <p>Complex Event Processing and supporting prediction components will allow to observe critical parameters of the subsystems and prepare/perform actions before the critical events occur.</p>

6 Conclusion

So far we have presented a view-based design time and runtime architecture for virtual service platforms (VSP). We elaborated on how the notion of architectural

views have been exploited for dealing with the complexity of the horizontal dimension, i.e., the dimension of different concerns, of a VSP, and how the model-driven development paradigm is leveraged for the separation of abstraction levels. We also proposed a number of view models that can be used for formalising essential VSP concerns such as the Service Component View, Deployment View, and Runtime View.

In order to provide view models that are more appropriate and relevant to the various stakeholders' interests, we devised a model-driven stack that organises these view models into abstract and technology-specific layer. The abstract layer includes view models that offer high-level concepts and structures of which non-technical stakeholders can better understand and communicate to discuss on certain business goals or requirements. The technology-specific layer consists of view models that are merely relevant to developers who are responsible for implementing, deploying, and maintaining VSPs. This combination of the *separation of concerns principle* and the *separation of abstraction levels* offers a flexible, extensible methodology for VSP development. Furthermore, the view-based architecture can also support code generator technique for generating VSP code, deployment configurations, monitoring directives, etc., from views, and therefore, enhances the automation and productivity.

We also proposed future plans for the potential integration of the view-based architecture with other components of INDENICA. The integration is naturally based on the collaboration between WP3 and other WPs in the upcoming milestones.

Table of Figures

Figure 1 Relationships with other INDENICA components.....	7
Figure 2 Overview of an INDENICA Virtual Service Platform.....	8
Figure 3: the technical architecture overview diagram from the EAM methodology with sample components in the respective layers	9
Figure 4 Overview of the view-based design time and runtime architecture.....	10
Figure 5 The Core model.....	12
Figure 6 The high-level Service Component View model.....	13
Figure 7 Essential OCL-based rules for validating a Service Component View	14
Figure 8 An example of the Service Component View.....	15
Figure 9 The Service Deployment View.....	15
Figure 10 Essential OCL-based rules for validating the Service Deployment View	16
Figure 11 An example of the Service Deployment View	17
Figure 12 An example of an SCA-based system.....	18
Figure 13 The low-level Service Component View for SCA technology	20
Figure 14 An example of the low-level Service Component View for SCA	21
Figure 15 Refining non-nested high-level components to low-level components.....	22
Figure 16 Refining nested high-level components to low-level components.....	23
Figure 17 The Runtime View model	24
Figure 18 An example of Runtime View in textual syntax.....	25
Figure 19 Overview of different roles of stakeholders in INDENICA architecture	29
Figure 20 NIST enterprise architecture model (http://en.wikipedia.org/wiki/File:NIST_Enterprise_Architecture_Model.jpg)	33
Figure 21: The Siemens Enterprise Architecture Management framework	35
Figure 22: INDENICA architecture views mapped to the planning phase of the EAM development cycle	35
Figure 23: Runtime Model Extension Capturing Monitoring Rules	36

References

- [CBB+2010] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., et al. (2010). *Documenting Software Architectures: Views and Beyond* (2nd ed., p. 592). Addison-Wesley.
- [DoW] INDENICA Project Description of Work, April 23th, 2010.
- [DR2011] Davis, Rob (2011). *Processes in Practice: Putting the "E" back into Enterprise Architecture*. <http://www.bptrends.com/>
- [DS2005] Dustdar, S., & Schreiner, W. (2005). A survey on web services composition. *International Journal of Web and Grid Services*, 1(1), 1-30. Inderscience. doi:10.1504/IJWGS.2005.007545
- [EMF] Eclipse Modeling Framework. <http://www.eclipse.org/modeling/emf>
- [FEM+2007] F. Rosenberg, C. Enzi, A. Michlmayr, C. Platzer, and S. Dustdar. Integrating Quality of Service Aspects in Top- Down Business Process Development Using WS-CDL and WS-BPEL. In EDOC '07: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference, page 15, Washington, DC, USA, 2007. IEEE Computer Society.
- [GEF] Eclipse Graphical Editing Framework. <http://www.eclipse.org/gef>
- [GMF] Eclipse Graphical Modeling Framework. <http://www.eclipse.org/gmf>
- [HC2001] G. T. Heineman and W. T. Councill, Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley, 2001.
- [JB2005] A. Jansen and J. Bosch, "Software Architecture as a Set of Architectural Design Decisions," in The 5th Working IEEE/I- FIP Conf. Software Architecture, pp. 109–120, IEEE Comp. Soc., 2005.
- [LK2007] L. Lee and P. Kruchten, "Capturing Software Architectural Design Decisions," in 2007 Canadian Conference on Electrical and Computer Engineering, pp. 686–689, IEEE, 2007.
- [LW2007] K.-K. Lau and Z. Wang, "Software Component Models," IEEE Trans. Softw. Eng., vol. 33, no. 10, pp. 709–724, 2007.
- [M2T] Eclipse M2T Project. <http://www.eclipse.org/modeling/m2t>
- [MM2004] Milanovic, N., & Malek, M. (2004). Current Solutions for Web Service Composition. *IEEE Internet Computing*, 8(6), 51-59. doi:10.1109/MIC.2004.58
- [OMG-UML] OMG. Unified Modeling Language. <http://www.omg.org/spec/UML/2.0>
- [OSOA] Open SOA (2007). Service Component Architecture (SCA) Specifications V1.00. <http://www.osoa.org>
- [OSGi] OSGi Alliance. Open Services Gateway initiative framework. <http://www.osgi.org>

-
- [OZD2010] E. Oberortner, U. Zdun, S. Dustdar, Patterns for Measuring Performance-Related QoS Properties in Distributed Systems. In Proceedings of the Pattern Languages of Programming Conference 2010 (PLOP 2010), Reno, Nevada, USA, ACM, October, 2010.
- [RW2005] Rozanski, N., & Woods, E. (2005). *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives* (p. 576). Addison-Wesley Professional.
- [SLA] The Service Level Agreement Zone. <http://www.sla-zone.co.uk>
- [SR2007] Sessions, Roger (2007). *A Comparison of the Top Four Enterprise-Architecture Methodologies*. <http://msdn.microsoft.com/en-us/library/bb466232.aspx>
- [SV2006] T. Stahl and M. Völter. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006.
- [Szyperski2002] C. Szyperski, Component Software: Beyond Object-Oriented Programming. Boston, MA, USA: Addison-Wesley, 2nd ed., 2002.
- [TA2005] J. Tyree and A. Akerman, "Architecture Decisions: Demystifying Architecture," IEEE Softw., vol. 22, no. 2, pp. 19–27, 2005.
- [THZ+2009] Tran, H., Holmes, T., Zdun, U., & Dustdar, S. (2009). *Modeling Process-Driven SOAs - a View-Based Approach*. In J. Cardoso & W. M. P. van der Aalst (Eds.), Handbook of Research on Business Process Modeling. IGI Global.
- [TMD2009] Taylor, R. N., Medvidovic, N., & Dashofy, E. (2009). *Software Architecture: Foundations, Theory, and Practice* (p. 712). Wiley.
- [W3C] W3C. Web Service Definition Language 1.1. <http://www.w3.org/TR/wsdl>
- [Xtext] Eclipse Xtext Project. <http://www.xtext.org>
- [ZGK+2007] O. Zimmermann, T. Gschwind, J. Küster, F. Leymann, and N. Schuster, "Reusable architectural decision models for enterprise application development," in Proc. of QoSA 2007, pp. 15–32, Springer-Verlag, 2007.

Appendix

A. Formal grammar for the QoS language implemented in Xtext 2.0

```

grammar dsl.Runtime with org.eclipse.xtext.common.Terminals

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

generate runtime 'http://cs.univie.ac.at/swa/viewbased/runtime'

RuntimeView:
  {RuntimeView}
  name=QualifiedName
  (
    'measurement' '{'
      (measurement+=Measurement)*
    '}'
  )?
  (
    'actions' '{'
      (action+=Action)*
    '}'
  )?
  (
    'Service Level Objective'
    '{'
      (serviceLevelObjective+=ServiceLevelObjective)*
    '}'
  )?
  ;

Measurement:
  ResponseTime | ProcessingTime | AverageThroughput | Availability
  ;

ProcessingTime:
  {ProcessingTime}
  name=ID
  'ProcessingTime' expression=Expression
  ;

ResponseTime:
  {ResponseTime}
  name=ID
  'ResponseTime' expression=Expression
  ;

AverageThroughput:
  {AverageThroughput}
  name=ID
  'AverageThroughput' expression=Expression
  ;

Availability:
  {Availability}
  name=ID
  'Availability' expression=Expression
  ;

Expression:

```

```

{Expression}
predicate=PREDICATE value=VALUE unit=UNIT
;

enum UNIT:
  hour='hour' |
  minute='minute' |
  second='second' |
  millisecond='millisecond' |
  week='week' |
  day='day' |
  month='month' |
  year='year' |
  percent='%' |
  dollar='$' |
  euro='€'
;

terminal VALUE returns ecore::EDouble:
  ('0'..'9')* ( '.' ('0'..'9')+ )?
;

enum PREDICATE:
  Equal='=' |
  GreaterThan='>' |
  LessThan='<' |
  GreaterThanOrEqual='>=' |
  LessThanOrEqual='<=' |
  NotEqual='!=' |
  Not='not'
;

ServiceLevelObjective:
  {ServiceLevelObjective}
  name=ID
  '{'
  ('obliged='obliged=STRING)?
  'validity' 'start'=DATETIME 'end'=end=DATETIME
  'measurement=' measurement+=[Measurement] ( ','
measurement+=[Measurement])*
  'action=' action+=[Action] ( ',' action+=[Action])*
  'target=' target+=ID ( ',' target+=ID)*
  '}'
;

Action:
  AdaptationAction | NotificationAction
;

AdaptationAction:
  ChangeServiceBinding | SwitchVariant | RegenerateAndDeploy
;

ChangeServiceBinding:
  name=ID 'ChangeServiceBinding'
;

RegenerateAndDeploy:
  name=ID 'RegenerateAndDeploy'
;

```

```

SwitchVariant:
  name=ID 'SwitchVariant'
  ;

NotificationAction:
  SendMail | SendSMS | Log
  ;

SendMail:
  {SendMail}
  name=ID
  'Send e-mail to'email+=STRING ( ',' email+=STRING)*
  ;

SendSMS:
  {SendSMS}
  name=ID
  'Send SMS to'sms+=SMS ( ',' sms+=SMS)*
  ;

Log:
  {Log}
  name=ID
  'Log event to file'file+=STRING ( ',' file+=STRING)*
  ;

terminal SMS:
  ('+')? ('0'..'9')+
  ;

terminal DATETIME:
  ('1'..'2') ('0'..'9') ('0'..'9') ('0'..'9') // YYYY
  '-' ('0'..'1') ('0'..'9') // MM
  '-' ('0'..'3') ('0'..'9')? // DD
  ('T'
  ('0'..'2') ('0'..'9') // hh
  ':' ('0'..'5') ('0'..'9')? // mm
  ':' ('0'..'5') ('0'..'9')? // ss
  )?
  (('Z'| ('+'| '-' ('0'..'2') ('0'..'9') ':' ('0'..'5') ('0'..'9'))
  )? // UTC
  ;

QualifiedName:
  ID ('.' ID)*
  ;

```

B. Model-to-model transformation rules for refining view models

```

create sca::ScaView this
mapMultipleSingleComponentsToComposite(component::ComponentView cv):
  let components = cv.element.typeSelect(component::Component) :
  let connectors = cv.element.typeSelect(component::Connector) :
  resetGlobalIndex()
  -> storeGlobalVar("connectors", connectors)
  -> this.setName(cv.name)
  -> this.setId(cv.id)
  -> this.annotation.addAll(cv.annotation.duplicate())

```



```

-> this.element.add(mapComposite(components, cv.name))
-> this.element.addAll(connectors.createScaWire())
;

private create sca::ScaComposite this
mapComposite(List[component::Component] components, String name):
  this.setName(name)
  -> this.setTargetNamespace("http://" + this.name.toLowerCase())
  -> this.nestedComponent.addAll(components.mapComponent(this))
;

private create sca::ScaComponent this
mapComponent(component::Component c, sca::ScaComposite parent):
  let provideds = c.port.select(p|p.kind ==
component::PortKind::PROVIDED) :
  let requires = c.port.select(p|p.kind ==
component::PortKind::REQUIRED) :
  this.setName(c.name)
  -> this.setId(c.id)
  -> this.annotation.addAll(c.annotation.duplicate())
  -> this.port.addAll(provideds.mapProvidedPortToComponentService())
  ->
this.port.addAll(requires.mapRequiredPortToComponentReference())
  -> parent.port.addAll(provideds.select(p|isNotConnectedPort(p,
(List)getGlobalVar("connectors"))).mapProvidedPortToCompositeService(
this))
  -> parent.port.addAll(requires.select(p|isNotConnectedPort(p,
(List)getGlobalVar("connectors"))).mapRequiredPortToCompositeReferenc
e(this))
;

private create sca::ScaCompositeService this
mapProvidedPortToCompositeService(component::Port p,
sca::ScaComponent promoted):
  let promoteds = promoted.port.select(s|s.name == p.name &&
sca::ScaComponentService.isInstance(s)):
  this.setName(promoted.name + "_" + p.name)
  -> this.annotation.addAll(p.annotation.duplicate())
  -> this.setKind(p.kind)
  -> if (promoteds != null && promoteds.size > 0) then
this.setPromote((sca::ScaComponentService)promoteds.get(0))
;

private create sca::ScaCompositeReference this
mapRequiredPortToCompositeReference(component::Port p,
sca::ScaComponent promoted):
  let promoteds = promoted.port.select(r|r.name == p.name &&
sca::ScaComponentReference.isInstance(r)):
  this.setName(promoted.name + "_" + p.name)
  -> this.annotation.addAll(p.annotation.duplicate())
  -> this.setKind(p.kind)
  -> if (promoteds != null && promoteds.size > 0) then
this.promote.add((sca::ScaComponentReference)promoteds.get(0))
;

private create sca::ScaComponentService this
mapProvidedPortToComponentService(component::Port p):
  let provideds = getGlobalVar("provideds") != null ?
getGlobalVar("provideds") : {} :
  this.setName(p.name)

```

```

-> this.annotation.addAll(p.annotation.duplicate())
-> this.setKind(p.kind)
-> provideds.add(this)
-> storeGlobalVar("provideds", provideds)
;

private create sca::ScaComponentReference this
mapRequiredPortToComponentReference(component::Port p):
  let requireds = getGlobalVar("requireds") != null ?
getGlobalVar("requireds") : {} :
  this.setName(p.name)
  -> this.annotation.addAll(p.annotation.duplicate())
  -> this.setKind(p.kind)
  -> requireds.add(this)
  -> storeGlobalVar("requireds", requireds)
;

private create sca::ScaWire this createScaWire(component::Connector
c):
  setName(c.name)
  -> this.annotation.addAll(c.annotation.duplicate())
  -> setSource(c.source.mapRequiredPortToComponentReference())
  -> setTarget(c.target.mapProvidedPortToComponentService())
;

create sca::ScaComposite this
mapSingleComponentToComposite(component::Component c):
  let promoted = c.mapSingleComponentToComponent() :
  let provided = c.port.select(p|p.kind ==
component::PortKind::PROVIDED) :
  let required = c.port.select(p|p.kind ==
component::PortKind::REQUIRED) :
  let services =
provided.mapProvidedPortToCompositeService(promoted) :
  let references =
required.mapRequiredPortToCompositeReference(promoted):
  this.setId(c.id)
  -> this.setName(c.name)
  -> this.annotation.addAll(c.annotation.duplicate())
  -> this.port.addAll(services)
  -> this.port.addAll(references)
  -> this.nestedComponent.add(promoted)
;

create sca::ScaComponent this
mapSingleComponentToComponent(component::Component c):
  let provided = c.port.select(p|p.kind ==
component::PortKind::PROVIDED) :
  let required = c.port.select(p|p.kind ==
component::PortKind::REQUIRED) :
  this.setName(c.name)
  -> this.annotation.addAll(c.annotation.duplicate())
  -> this.setId(c.id)
  -> this.port.addAll(provided.mapProvidedPortToComponentService())
  ->
this.port.addAll(required.mapRequiredPortToComponentReference())
;

```