Core meta-models, transformation templates, and languages

Project acronym: COMPAS
Project name: Compliance-driven Models, Languages, and Architectures for Services
Call and Contract: FP7-ICT-2007-1
Grant agreement no.: 215175
Project Duration: 01.02.2008 – 28.02.2011 (36 months)

Co-ordinator: TUV Vienna University of Technology (AT)
Partners:

CWI Stichting Centrum voor Wiskunde en Informatica (NL)
UCBL Université Claude Bernard Lyon I (FR)
USTUTT Universitaet Stuttgart (DE)
TILBURG UNIVERSITY Stichting Katholieke Universiteit Brabant (NL)
UNITN Universita degli Studi di Trento (IT)
TARC-PL Telecordia Poland (PL)
THALES Thales Services SAS (FR)
pWC Pricewaterhousecoopers Accountants N.V. (NL)

This project is supported by funding from the Information Society Technologies Programme under the 7th Research Framework Programme of the European Union.
Project no. 215175

COMPAS

Compliance-driven Models, Languages, and Architectures for Services

Specific Targeted Research Project
Information Society Technologies

Start date of project: 2008-02-01     Duration: 36 months

D1.2 Core meta-models, transformation templates, and languages

Revision 1.0

Due date of deliverable: 2009-12-31
First submission date: 2009-12-22
Latest submission date:

Organisation name of lead partner for this deliverable:
TUV Technische Universität Wien, AT

Contributing partner(s):
CWI Stichting Centrum voor Wiskunde en Informatica, NL
USUTT Universitaet Stuttgart, DE
TILBURG UNIVERSITY Stichting Katholieke Universiteit Brabant, NL
THALES Thales Services SAS, FR

| Project funded by the European Commission within the Seventh Framework Programme |
| Dissemination Level                                                                 |
| PU  | Public                                                                 |             |
| PP  | Restricted to other programme participants (including the Commission Services) | X            |
| RE  | Restricted to a group specified by the consortium (including the Commission Services) |             |
| CO  | Confidential, only for members of the consortium (including the Commission Services) |             |
History chart

<table>
<thead>
<tr>
<th>Issue</th>
<th>Date</th>
<th>Changed page(s)</th>
<th>Cause of change</th>
<th>Implemented by</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>2008-10-30</td>
<td>Table of contents</td>
<td>New document</td>
<td>TUV</td>
</tr>
<tr>
<td>0.2</td>
<td>2009-11-08</td>
<td>Section 2,3</td>
<td>Add content</td>
<td>TUV</td>
</tr>
<tr>
<td>0.3</td>
<td>2009-11-20</td>
<td>Section 5</td>
<td>Add content</td>
<td>TUV</td>
</tr>
<tr>
<td>0.4</td>
<td>2009-12-01</td>
<td>Section 4</td>
<td>Add content</td>
<td>TUV</td>
</tr>
<tr>
<td>0.5</td>
<td>2009-12-10</td>
<td>Whole document</td>
<td>Revise some places and fix typos</td>
<td>TUV</td>
</tr>
<tr>
<td>0.6</td>
<td>2009-12-15</td>
<td>Section 2,3,4,5</td>
<td>Revise according to collaboration with partners</td>
<td>TUV+USUTT+CWI+TILBURG+THALES</td>
</tr>
<tr>
<td>0.9</td>
<td>2009-12-17</td>
<td>Whole document</td>
<td>Finally revise for releasing</td>
<td>TUV</td>
</tr>
<tr>
<td>1.0</td>
<td>2009-12-22</td>
<td></td>
<td>Approval &amp; Release</td>
<td>TUV</td>
</tr>
</tbody>
</table>

Authorisation

<table>
<thead>
<tr>
<th>No.</th>
<th>Action</th>
<th>Company/Name</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Prepared</td>
<td>TUV</td>
<td>2009-12-17</td>
</tr>
<tr>
<td>2</td>
<td>Approved</td>
<td>TUV</td>
<td>2009-12-22</td>
</tr>
<tr>
<td>3</td>
<td>Released</td>
<td>TUV</td>
<td>2009-12-22</td>
</tr>
</tbody>
</table>

Disclaimer: The information in this document is subject to change without notice. Company or product names mentioned in this document may be trademarks or registered trademarks of their respective companies.

All rights reserved.

The document is proprietary of the COMPAS consortium members. No copying or distributing, in any form or by any means, is allowed without the prior written agreement of the owner of the property rights.

This document reflects only the authors’ view. The European Community is not liable for any use that may be made of the information contained herein.
## Contents

1. Introduction ................................................................................................................................. 8
   1.1. Purpose and scope................................................................................................................ 8
   1.2. Document overview............................................................................................................. 8
   1.3. Definitions and glossary...................................................................................................... 8
   1.4. Abbreviations and acronyms............................................................................................... 9

2. MDSD Software Framework overview ................................................................................... 11

3. View-based Modelling Framework (VbMF) .......................................................................... 13
   3.1. Fundamental concepts ....................................................................................................... 13
   3.2. View-based, model-driven approach for process-driven SOAs........................................ 16
   3.3. Modelling basic process concerns .................................................................................... 18
      3.3.1. Core model.................................................................................................................. 18
      3.3.2. FlowView model ........................................................................................................ 19
      3.3.3. CollaborationView model .......................................................................................... 22
      3.3.4. InformationView model ............................................................................................. 24
   3.4. Modelling additional process concerns ............................................................................ 25
      3.4.1. HumanView model .................................................................................................... 25
      3.4.2. TransactionView model ............................................................................................. 26
      3.4.3. EventView model ....................................................................................................... 27
   3.5. View manipulation mechanisms ....................................................................................... 28
      3.5.1. Extension mechanism ................................................................................................ 28
      3.5.2. Integration mechanism ............................................................................................... 32
      3.5.3. Code generation .......................................................................................................... 34
   3.6. View-based reverse engineering ....................................................................................... 35
      3.6.1. View-based reverse engineering approach ............................................................... 36
      3.6.2. General approach for view extraction ....................................................................... 37
      3.6.3. Recovering abstract, high-level representations ....................................................... 38
      3.6.4. Recovering low-level, technology-specific representations .................................... 40
   3.7. Tool support ....................................................................................................................... 41
      3.7.1. View-based, model-driven forward engineering ...................................................... 41
      3.7.2. View-based reverse engineering ............................................................................... 45
   3.8. Summary ............................................................................................................................ 46

4. Domain-specific languages in COMPAS ................................................................................ 46
   4.1. DSL development process................................................................................................. 47
      4.1.1. Language model driven DSL development .............................................................. 48
4.1.2. Mockup-language driven DSL development ............................................................ 50
4.1.3. Extracting the DSL from an existing system ............................................................ 50
4.2. Concrete syntax style ......................................................................................................... 51
  4.2.1. Textual concrete syntax ............................................................................................. 51
  4.2.2. Textual concrete syntax with generation of visualisations ...................................... 52
  4.2.3. Graphical concrete syntax ........................................................................................ 52
  4.2.4. Form- or Table-based Concrete Syntax .................................................................... 52
4.2. Concrete syntax style ......................................................................................................... 51
  4.2.1. Textual concrete syntax ............................................................................................. 51
  4.2.2. Textual concrete syntax with generation of visualisations ...................................... 52
  4.2.3. Graphical concrete syntax ........................................................................................ 52
  4.2.4. Form- or Table-based Concrete Syntax .................................................................... 52
4.3. External vs. embedded DSL .......................................................................................... 53
  4.3.1. External DSL .............................................................................................................. 53
  4.3.2. Embedded DSL .......................................................................................................... 53
4.4. DSLs for compliance concerns ...................................................................................... 55
  4.4.1. The language models.................................................................................................. 56
  4.4.2. Implementation ........................................................................................................... 58
  4.4.3. Using the DSLs........................................................................................................... 59
  4.4.4. Generated code ........................................................................................................... 61
4.5. Summary ............................................................................................................................ 62
5. Prototype Integration ............................................................................................................ 62
  5.1. Introduction ....................................................................................................................... 62
  5.2. Compliance Metadata model ........................................................................................ 64
  5.3. Integration of related prototypes ................................................................................... 66
6. Conclusions ........................................................................................................................... 66
7. Reference documents ........................................................................................................... 67
  7.1. Internal documents ......................................................................................................... 67
  7.2. External documents ....................................................................................................... 67
List of figures

Figure 1  The MDSD Software Framework in overall COMPAS architecture [D1.1] ...... 11
Figure 2  Interaction and integration of COMPAS technologies and prototypes [D1.1] .... 12
Figure 3  Fundamental concepts of the view-based, model-driven approach .................. 13
Figure 4  Layered architecture of the view-based, model-driven approach .................... 15
Figure 5  The Ecore meta-model [SBP+08] – an MOF-compliant meta-model .............. 16
Figure 6  View-based modelling framework architecture ............................................... 17
Figure 7  The Core model .............................................................................................. 18
Figure 8  The Core model of the Travel Booking process .............................................. 19
Figure 9  The FlowView model .................................................................................. 20
Figure 10 The FlowView of the Travel Booking process ............................................... 22
Figure 11 The CollaborationView model ....................................................................... 23
Figure 12 The CollaborationView and InformationView of the Travel Booking process ... 24
Figure 13 The InformationView model ......................................................................... 25
Figure 14 The HumanView model ................................................................................ 26
Figure 15 The TransactionView model .......................................................................... 27
Figure 16 The EventView model .................................................................................... 28
Figure 17 Illustration of VbMF extensibility based on the Core model ............................ 29
Figure 18 BpelInformationView model - an extension of the InformationView model ...... 30
Figure 19 A BpelCollaboration and BpelInformation View of the Travel Booking process 31
Figure 20 Name-based matching algorithm .................................................................. 33
Figure 21 View-based model-to-code transformation (code generation) ....................... 34
Figure 22 Code generation template rules and the generated schematic BPEL code ........ 35
Figure 23 Overview of the view-based reverse engineering approach ............................. 36
Figure 24 Recovering the FlowView model from BPEL descriptions ............................ 38
Figure 25 Recovering the CollaborationView from WSDL descriptions ......................... 39
Figure 26 Recovering the CollaborationView and BpelCollaborationView ..................... 40
Figure 27 View-based, model-driven development tool chain ....................................... 41
Figure 28 The proof-of-concept tooling of the view-based, model-driven approach ......... 43
Figure 29 View-based reverse engineering tool chain for process-driven SOAs .............. 45
Figure 30 Design space overview ................................................................................ 47
Figure 31 Decision overview: DSL development process .............................................. 48
Figure 32 Language model driven DSL engineering process ........................................ 49
Figure 33 Decision overview: Concrete syntax style ..................................................... 51
Figure 34 Decision overview: External vs. Embedded DSL ........................................ 53
Figure 35  Our incremental DSL development approach ........................................................ 54
Figure 36  An example of annotating processes or services with QoS compliance concerns56
Figure 37  Extending the high-level QoS language model with the additionally needed technical details ............................................................................................................................. 57
Figure 38  The definition of the high-level domain model using Frag ................................. 58
Figure 39  The definition of the low-level domain model using Frag ................................. 59
Figure 40  The high-level QoS DSL editor ......................................................................... 60
Figure 41  View-based approach for modelling processes and business compliance .......... 64
Figure 42  Proposed solution for integrating data models of COMPAS prototypes .......... 65
Figure 43  Integration of related prototypes ....................................................................... 66

List of tables
Table 1  View-based modelling framework components ......................................................... 17
Table 2  The semantics of basic control structures of the FlowView model ......................... 21

List of listings
Listing 1  A workflow for generating process code and deployment configurations .......... 44
Listing 2  Using the low-level QoS DSL .............................................................................. 60
Listing 3  A generated interceptor for measuring the processing time of a service .......... 61
Abstract

Ensuring business compliance in process-driven SOAs is a tedious and error-prone task because the stakeholders confront two challenges: the increasing complexity of process descriptions and the gap between abstraction levels due to the difference of language syntax and semantics, the difference of granularity, and the lack of supporting links between high-level and low-level process languages. In this deliverable, we present a MDSD Software Framework comprising a view-based, model-driven approach and various DSLs tooling to address the aforementioned issues in order to ease the process development in a flexible, extensible manner. The MDSD Software Framework will contribute a foundation for modelling and development processes and business compliance as well as provide appropriate means for integrating COMPAS partners’ efforts at both conceptual and technical levels.

1. Introduction

1.1. Purpose and scope

Within the overall scope of the COMPAS project, WP1 focuses on development of the core and modelling aspects of this business compliance framework as well as ensuring that it integrates with the components from the different WPs in COMPAS. The purpose of this deliverable is to elaborate the MDSD Software Framework – which is the foundation for modelling and development of business processes and compliance. In addition, the high-level integration architecture mentioned in [D1.1] is refined and explained in detail how this framework integrates with related WPs.

1.2. Document overview

This document is organized as following. Section 2 recalls the high level overview of COMPAS architecture mentioned in [D1.1] and highlights the MDSD Software Framework and its relationships to other components. Next, Section 3 presents the view-based, model-driven approach and a realisation of this approach in terms of a View-based Modelling Framework. Domain-specific languages used for formulating domain and compliance concepts and techniques are presented in Section 4. Finally, Section 5 comes to clarify the integration between the MDSD Software Framework and related components from other WPs.

1.3. Definitions and glossary

Separation of concerns: the process of breaking a software system into distinct pieces such that the overlaps between those pieces are as little as possible, in order to make it easier to understand, design, develop, maintain, etc., the system.

Domain-Specific Languages (DSLs): DSLs are languages that are specifically tailored for a particular domain. Usually, DSLs are simple because they are suited for a very narrow purpose only, and they are easy to edit and to translate. To describe a broad domain, a broad DSL can be used. To keep the smallness and simplicity of DSLs, multiple narrow DSLs should be used, which have to be combined to describe a broad domain. The goal of DSLs is to improve productivity and software quality. DSLs raise the level of abstraction to empower users with the
ability to build solutions using concepts that are similar to the domain and his/her knowledge.

*Model-Driven Software Development (MDSD)* or *Model-Driven Development (MDD)*: a paradigm that advocates the concept of models, that is, models will be the most important development artefacts at the centre of developers’ attention. In MDSD, domain-specific languages are often used to create models that capture domain abstraction, express application structure or behaviour in an efficient and domain-specific way. These models are subsequently transformed into executable code by a sequence of model transformations.

*Model transformation*: transformation maps high-level models into low-level models (aka model-to-model transformations), or maps models into source code, executable code (aka model-to-code or code generation).

*Stakeholders*: In general, stakeholder is a person or organization with a legitimate interest in a given situation, action or enterprise. In the context of this chapter, stakeholder is a person who involved in the business process development at different levels of abstraction, for instance, the business experts, system analysts, IT developers, and so forth.

### 1.4. Abbreviations and acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPEL</td>
<td>Business Process Execution Language</td>
</tr>
<tr>
<td>BPMN</td>
<td>Business Process Modelling Notation</td>
</tr>
<tr>
<td>BPML</td>
<td>Business Process Modelling Language</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain Specific Language</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modelling Framework</td>
</tr>
<tr>
<td>EPC</td>
<td>Event-Driven Process Chain</td>
</tr>
<tr>
<td>FDL</td>
<td>WebSphere FlowMark Definition Language</td>
</tr>
<tr>
<td>jPDL</td>
<td>jBPM Process Definition Language</td>
</tr>
<tr>
<td>M2C</td>
<td>Model-to-code transformation</td>
</tr>
<tr>
<td>M2M</td>
<td>Model-to-model transformation</td>
</tr>
<tr>
<td>MDA</td>
<td>Model-Driven Architecture</td>
</tr>
<tr>
<td>MDSD</td>
<td>Model-Driven Software Development</td>
</tr>
<tr>
<td>OAW</td>
<td>openArchitectureWare</td>
</tr>
<tr>
<td>OCL</td>
<td>Object-Constraint Language</td>
</tr>
<tr>
<td>POJO</td>
<td>Plan Old Java Object</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>RBAC</td>
<td>Role-based Access Control</td>
</tr>
<tr>
<td>SOA</td>
<td>Service-Oriented Architecture</td>
</tr>
<tr>
<td>VbMF</td>
<td>View-based Modelling Framework</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Services Description Language</td>
</tr>
<tr>
<td>WSBPEL</td>
<td>Web Services Business Process Execution Language</td>
</tr>
<tr>
<td>XPDL</td>
<td>XML Process Definition Language</td>
</tr>
</tbody>
</table>
2. MDSD Software Framework overview

In [D1.1] an overall COMPAS architecture has been presented including various design time and runtime components. A very important part of the COMPAS design time architecture is the MDSD Software Framework highlighted in Figure 1 and Figure 2. The MDSD Software Framework is made up by a View-based Modelling Framework (VbMF) [TZD07,TZD08a,TZD08b,HTZ+08,THZ+09] that provides the basis for modelling and developing SOAs and process-driven SOAs along with DSLs tooling for describing and formulating compliance concerns such as security, licensing, business policies, QoS, and so on, and/or other supporting elements.

As mentioned in [D1.1], these DSLs are means used by the stakeholders in their respective domains to model a particular organisation’s interpretation of compliance concerns. And VbMF provides means in terms of core view models and supporting mechanisms for modelling and developing these concerns in separate views, integrating views to achieve a richer or a more thorough view, and generating process-driven SOAs implementation and runtime supporting configurations (see Figure 2).

In the subsequent sections, we will elaborate the View-based Modelling Framework and the DSLs tooling that constitute the MDSD Software Framework. We also present relevant support for integrating the MDSD Software Framework with prototypes from other WPs.
Figure 2 Interaction and integration of COMPAS technologies and prototypes [D1.1]
3. View-based Modelling Framework (VbMF)

3.1. Fundamental concepts

A typical business process embodies various tangled concerns, such as the control flow, data processing, service and process invocations, fault handling, event handling, human interactions, transactions, to name but a few. The entanglement of those concerns increases the complexity of process development and maintenance as the number of involved services and processes grow. In order to deal with this complexity, we use the notion of architectural views (or views for short) to describe the various process concerns. In particular, a view is a representation of one particular concern of a process. We devise different view models for formalising the concept of architectural view. A view model specifies the abstract syntax, i.e., the view’s structure, and the static semantics, i.e., the meaning of the view’s structure which a view derived from that view model must conform to. The view models, in turn, are defined on top of the meta-model. Figure 3 shows the fundamental concepts of our view-based, model-driven approach along with their relationships to each other [TZD07,THZ+09].

![Figure 3](image_url)

**Figure 3 Fundamental concepts of the view-based, model-driven approach**

In Figure 3, we present the formalisations of basic process concerns such as the control flow, service invocations, and data processing, in terms of the FlowView, CollaborationView, and
Information View model, respectively. These view models, which are presented in detail in the subsequent sections, are built up around a Core model. The Core model is intentionally developed for conceptually representing the essence of a business process. That is, the Core model covers three distinct concepts: the process, the relationships between process and the environment, i.e., the services, and the internal representation of the process, i.e., the process views. Process concerns described by the view models merely relate to these concepts in the sense that each concern involves either the process’s interior or exterior, or both. In other words, the other view models derive and extend the foundational concepts provided in the Core model as shown in Figure 3. As a result, the Core model plays an important role in our approach because it provides the basis for extending and integrating view models, and establishing and maintaining the dependencies between view models [TZD07, THZ+09].

Nonetheless, our view-based approach is not limited to these concerns, but can be extended to cover various other concerns. For instance, human interactions, transactions, event handling have been realised as extensions [TZD07, HTZ+08, THZ+09]. A new concern can be integrated into our approach by using 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 process concerns, we can extend the view-based approach along the horizontal dimension, i.e., the dimension of process concerns, to deal with the complexity caused by the various tangled process concerns. There are many stakeholders involved in process development at different levels of abstraction. For instance, business experts require high-level abstractions that offer domain or business concepts concerning their distinct knowledge, skills, and needs, while IT experts merely work with low-level, technology-specific descriptions. The MDD paradigm [SV06] 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 MDD paradigm, we devise a model-driven stack that has two basic layers: abstract and technology-specific. The abstract layer includes the views without the technical details such that the business experts can understand and manipulate. Then, the IT experts can refine or map these abstract concepts into platform- and technology-specific views.
The technology-specific layer contains the views that embody concrete information of technologies or platforms. On the one hand, a technology-specific view model can be directly derived from the Core model, such as the TransactionView model shown in Figure 4. One the other hand, a technology-specific view model can also be an extension of an abstract one, for instance, the BpelCollaborationView model extends the CollaborationView model, the BPEL PeopleView model extends the HumanView model, etc., by using the model refinement mechanism. By refining an abstract layer down to a technology-specific layer, our view-based approach helps bridging the abstraction levels along the vertical dimension, i.e., the dimension of abstraction, which is orthogonal to the horizontal dimension described in the previous paragraph [TZD07,THZ+09].

According to the specific needs and knowledge of the stakeholders, views can be combined to provide a richer view or a more thorough view of a certain process. For instance, IT experts may need to involve the process control flow along with service interactions which is only provided via an integration of the FlowView with either the CollaborationView or BpelCollaborationView [TZD07,THZ+09].

Based on the aforementioned view model specifications, stakeholders can create different types of views for describing specific business processes. These process views can be instances of the concerns’ view models, extension view models, or integrated view models (see Figure 3). They can be manipulated by the stakeholders to achieve a certain business goal, or adapt to new requirement in business environment or changes in technology and platform. Finally, we provide model-to-code transformations (or so-called code generations) that take these views as inputs and generate process implementations and deployment configurations. The resulting code and configurations, which may be augmented with hand-written code, can be deployed in process engines for execution.
In the subsequent sections, we present in detail the concepts and mechanisms described above as well as the realisation of these concepts in our prototype, namely, the View-based Modelling Framework (VbMF).

3.2. View-based, model-driven approach for process-driven SOAs

![Diagram of Ecore meta-model](image)

Figure 5 The Ecore meta-model [SBP+08] – an MOF-compliant meta-model

The meta-model, which is the basis of our view models, can be simple or more elaborate like Meta-Object Facility (MOF) [MOF]. For better integration and interoperability with existing modelling and development tools which are using MOF-compliant models and utilising the XMI standard [XMI] for persisting models, we choose the Eclipse Modelling Framework (EMF) [EMF] for realising our view-based, model-driven approach. Figure 5 shows an excerpt of the EMF Ecore meta-model which is used as the basis for deriving our view models.

Based on Ecore, we define the Core model as the foundation for the View-based Modelling Framework. After that, different view models are derived from the Core model to represent the various process concerns. In Figure 6, we present the overall architecture of VbMF in which modelling artefacts such as view models and views are created, manipulated, and maintained. We categorize distinct functional components of VbMF as shown in Table 1.

In the VbMF forward engineering tool chain, abstract views are designed first. Next, these abstract views are refined down to their lower level counterparts, which are technology-specific views. The code generator then uses the technology-specific views to produce schematic process code and/or the necessary configurations for deployment and execution. In the course of the reverse engineering tool chain, the view-based interpreters take as input legacy process descriptions and produce abstract or technology-specific views that can be reused later in the forward engineering tool chain.
Figure 6 View-based modelling framework architecture

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>View model editors</td>
<td>View model editors are based on the view models. Using these editors, a new view model can be developed from scratch by deriving the Core model. Moreover, an existing view model can also be extended with some additional features to form a new view model. We use the basic Ecore editor generator to develop our view model editors.</td>
</tr>
<tr>
<td>View editors</td>
<td>View editors can be (semi)-automatically generated from VbMF view models based on the editor generations provided in EMF. The view editors support stakeholders in creating new views or editing existing ones. For the sake of simplicity and illustration purposes, we choose the simplest visualisation of the view editors, which is a tree-view editor.</td>
</tr>
<tr>
<td>View integrators</td>
<td>View integrators aid the stakeholders in combining view models to produce a richer view or a more thorough view of a certain process</td>
</tr>
<tr>
<td>Code generators</td>
<td>Code generators generate schematic recurrent code and configurations from one or many views. Before generating outputs, the code generators will validate the conformity of the input views against the corresponding view models.</td>
</tr>
<tr>
<td>View-based interpreters</td>
<td>View-based interpreters are the key contributions of our view-based reverse engineering approach which can be used to extract appropriate views from existing process descriptions.</td>
</tr>
</tbody>
</table>

Table 1 View-based modelling framework components
3.3. Modelling basic process concerns

In this section, we present in detail the (semi-)formalized representations of basic process concerns in terms of view models, including the control flow, process interactions, and data handling concerns [TZD07,HTZ+08,THZ+09].

3.3.1. Core model

Aiming at the openness and the extensibility, we devised the Core model as a foundation for creating the other view models (see Figure 7). The Core model comprises a number of conceptual classes: View, Process, and Service. Each of these classes can be extended further. The Core model conforms to the Ecore meta-model shown in Figure 5. In particular, classes shown as rectangle shapes, for instance, NamedElement, Namespace, Element, View, Service, and Process, are instances of EClass. The attributes of a class, for instance, name and nsURI of the NamedElement class, are instances of EAttribute, and the relationships between two classes are instances of EReference.

At the heart of the Core model is the View class that represents the concept of architectural view. Each specific view (i.e., a specialization of the View class) represents one perspective of a Process. A Service stands for functionality that the corresponding Process provides or requires. The meaning of Process is self-explained. A View also stands for a container of several Elements representing the objects that describe process interior. Different instances of each of those classes can be distinguished through the features of the common supertype NamedElement, defining a name property (name) and a namespace URI (nsURI). Figure 8 depicts a Core model of the Travel Booking process described in [IBM06].

---

**Figure 7 The Core model**

Diagram showing the relationships between the CoreModel, Service, Process, View, NamedElement, and Element classes.
View models representing various concerns of a business process 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 not only for extension and dependency management, but also for defining the integration points used to merge view models as mentioned in the description of the integration mechanisms below.

### 3.3.2. FlowView model

As we mentioned above, the notion of process is central in process-driven SOAs. Therefore, the flow of control is one of the most important concerns of a process which is formalised by a FlowView model. A FlowView model embodies many activities and control structures. The activities are process tasks such as service invocations or data handling, while control structures describe the execution order of the activities in order to achieve a certain goal.
There are several approaches to modelling process control flows such as state-charts, block structures, UML Activity Diagram extensions, Petri-nets, and so on. Despite of this diversity in control flow modelling, it is widely accepted that existing modelling languages have the following patterns in common: Sequence, Parallel Split, Synchronization, Exclusive Choice, Simple Merge, and Structured Loop. Furthermore, zur Mühlen and Recker report a statistical result in how modelling elements are commonly used in practice [zR08]. The statistics indicates only a few of control structures are regularly used by process modellers in reality, which is accordant with the above observation of the common patterns. Therefore, we adopt these patterns as the building blocks of the FlowView model. Other, and more advanced, patterns can be added later by using extension mechanisms to augment the FlowView model.

We define the semantics of the control structures in the FlowView model with respect to these patterns in Table 2.

The primary class of the FlowView is the Task, which is the base class for other classes such as Sequence, Parallel, and Exclusive. Another important entity in the FlowView is the AtomicTask a specialization of Task that represents a single concrete action, such as a service invocation, a data processing task, etc. The actual description of each AtomicTask is only given in another specific view rather than in the FlowView. For instance, a service invocation is described in a CollaborationView, while a data processing action is specified in an InformationView. In other words, an AtomicTask is a placeholder or a reference to a specific activity. It is an interaction or a data processing task, which is defined respectively in the corresponding concern’s view. Therefore, every AtomicTask becomes an integration point that can be used to merge a FlowView with another view, for instance, an InformationView or a CollaborationView. The CompositeTask is an abstract representation of a group of related activities.

<table>
<thead>
<tr>
<th>Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>An activity is only enabled after the completion of another activity in the same Sequence structure. The Sequence structure is therefore equivalent to the semantics</td>
</tr>
</tbody>
</table>
of the Sequence pattern described in [vtK+03]

<table>
<thead>
<tr>
<th>Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td>All activities of a Parallel structure are executed in parallel. The subsequent activity of the Parallel structure is only enabled after the completion of all activities in the Parallel structure. The semantics of the Parallel structure is equivalent to a control block starting with the Parallel Split pattern and ending by the Synchronization pattern presented in [vtK+03]</td>
</tr>
<tr>
<td>Exclusive</td>
<td>Only one of many alternative paths (Branches or Default) of control inside an Exclusive structure is enabled according to a condition value of each branch. After the active branch finished, the process continues with the subsequent activity of the Exclusive structure. The semantics of the Exclusive structure is equivalent to a control block starting with the Exclusive Choice pattern and ending by the Simple Merge pattern proposed in [vtK+03]</td>
</tr>
<tr>
<td>Loop</td>
<td>Loop executes the containing task zero or more times while a pre-defined condition still evaluates to true. The condition is evaluated before the first iteration of the Loop and is re-evaluated prior to each of subsequent iterations. Once the condition evaluates to false, the thread of control passes to the task immediately following the Loop.</td>
</tr>
</tbody>
</table>

Table 2 The semantics of basic control structures of the FlowView model

Sequence, Parallel, Exclusive, and Loop are straightforward realizations of the patterns described in Table 2. Both Sequence and Parallel consist of unlimited number of process tasks which are executed sequentially or concurrently. A Sequence or a Parallel completes when all of the enclosed tasks have finished. An Exclusive structure consists of one or many Branches, each of which embodies a condition that decides the path of execution at run-time, and references to a particular Task being executed if the corresponding condition is satisfied. A Default may be included in an Exclusive structure in which the exceptional conditions that are not covered by any Branch's condition are handled. A Loop, which is a specialisation of Task, can be used for describing an iteration of a certain process task under some conditions. In Figure 10, we illustrate a FlowView that represents the control flow of the Travel Booking process.
3.3.3. CollaborationView model

A business process is often developed by composing the functionality provided by various parties, such as services or other processes. Other partners, in turn, may use the process. All business functions required or provided by the process are typically exposed in terms of standard interfaces (e.g., WSDL portTypes). So far, we have modelled these concepts in the Core model by the relationships between the two classes Process and Service. The CollaborationView model extends these concepts to represent the interactions between the process and its partners (see Figure 11).

Figure 10 The FlowView of the Travel Booking process
In the CollaborationView model, the Service of the Core model is extended by a tailored and specialized Service class that exposes a number of Interfaces. Each Interface provides some Operations, each of which stands for a particular functionality provided by the associated Service. The provided attribute of a Service is either true if that Service is provided by the process or false otherwise.

Each party involved in the process is described by a Partner class. The ability and the responsibility of a process's Partner are modelled by the associated Role. Functionality provided by a Partner is modelled by the Interface associated with a certain Role of the Partner. Collaboration between the process and a Partner is represented by an Interaction linking to that Partner. The type of communication in each Interaction is input, output, or both. In addition, an Interaction has a createsInstance attribute that indicates whether the Interaction triggers the execution of the process or not. This feature is needed to inform the process engine when to instantiate and start executing the process. It serves as the entry point for the process just like the main() method in programming languages such as C/C++ and Java. Last but not least, an Interaction task is performed whenever the process invokes a service or another process, or the process is invoked itself via its provided interface. Therefore, each Interaction is associated with a specific service's Interface and Operation.

Partner, Role, Interface, Operation, and Interaction are subtypes of the Element from the Core model. It means that those are classes of the CollaborationView, which is one specific subtype of the process Views. Furthermore, those classes inherit the name attribute that can be used in the named-based matching algorithm described in Section 3.5.2 to integrate view models.

In summary, the relationships between a process and the external environment such as other processes and services are modelled by a number of interactions. Each of these interactions requires a specific role of a certain partner of the process. A partner's role is reified through the functionality provided in terms of services each of which may expose one or many
operations via its interface. We present a CollaborationView of the Travel Booking process in Figure 12(a).

![CollaborationView and InformationView](image)

**Figure 12** The CollaborationView and InformationView of the Travel Booking process

### 3.3.4. InformationView model

The third process concern we consider in the context of this chapter is data handling which is formalised by the InformationView model (see Figure 13). This view model involves the representation of data object flows inside the process and message objects travelling back and forth between the process and its partners.
In the InformationView model, the BusinessObject class, which may have a generic type, namely, Type, is the abstraction of any piece of information, for instance, a purchase order received from the customer, or a request sent to a banking service to verify the customer's credit card, and so forth. Each InformationView comprises of a number of BusinessObjects. Messages exchanged between the process and its partners or data flowing inside the process might go through some Transformations wherein input data is converted or extracted to yield new pieces of output data. The transformations are performed inside a DataHandling class. The source or the target of a certain transformation is represented by an ObjectReference that holds a reference to a particular BusinessObject.

To summarise, the InformationView model conceptualises the data that are either processed inside the process or exchanged between the process and its partners. The Information view is the subtype of the View class. Its elements described in the previous paragraphs specialise the Element class of the Core model. Figure 12(b) presents an InformationView of the Travel Booking process.

### 3.4. Modelling additional process concerns

To illustrate the extensibility of our approach along the horizontal dimension in order to cover newly added process concerns, we present in this section some extra view models developed to formalise long-running transaction, event handling, and human interaction concerns of processes.

#### 3.4.1. HumanView model

A process task's functionality is often accomplished by invoking external services. However, business processes might also involve human interactions; for instance, a customer submits a purchase order that triggers an order processing process, a manager accepts or declines a loan approval in a loan approval process, and so on. We call such process elements HumanTasks. HumanTasks, thus, are special process tasks that are performed by a person. HumanTasks may need certain input values as well as a task description and may produce some output data.
Besides the HumanTask as a special process element, the HumanView model shown in Figure 14 defines human roles and their relationships to the respective process and tasks. Roles are abstracting concrete users that may play certain roles. The HumanView thus establishes a role-based abstraction. This role-based abstraction can be used for role-based access control (RBAC) [FBK99]. RBAC, in general, is administered through roles and role hierarchies that mirror an enterprise's job positions and organisational structure. Users are assigned membership into roles consistent with a user's duties, competency, and responsibility.

Examples for different types of Roles are task owner, process supervisor, and escalation recipient. By binding, for instance, the role of a process supervisor to a process, RBAC can define that those users that are associated with this role may monitor the process execution. Similarly, the owner of a HumanTask may complete the task by sending results back to the process.

We can specify an activity as defined within a FlowView to be a HumanTask in the HumanView that is, for instance, bound to an owner, i.e., the person who performs the task. Likewise, process stakeholders can be specified for the process by associating them with the HumanView.

### 3.4.2. TransactionView model

The TransactionView model depicted in Figure 15 represents long-running transactions of processes. This view embodies concepts which are specific to the BPEL standards. A Scope specifies the behaviour context of one or group of process activities. In such a context, developers may define a CompensationHandler, which invokes necessary activities to compensate the functionality done by the activities associated with the corresponding scope, and a FaultHandler for dealing with occurring faults or exceptions. In addition, the TransactionView also consists of some Compensates that explicitly invokes a certain CompensationHandler of the linked Scope. For more details of the transaction handling concern, refer the BPEL standard [WSBPEL].
Figure 15  The TransactionView model

Scope, Compensate, CompensationHandler, and FaultHandler are subtypes of the Element of the Core model because these elements, independently developed though, need to be integrated into the FlowView model of a process to augment the transactional semantics for that process.

3.4.3. EventView model

The EventView model (see Figure 16) describes the process ability to manage and handle events. On the one hand, a process can be triggered by message-based events, for instance, call-backs from the process partners. This type of events is represented by the OnMessage class. On the other hand, the process or its activities is also activated by timing events, for instance, the purchase order process will wait for a manual payment from the customer for one week. We model this type of events by the OnAlarm class that allows the modellers to specify timing period via one of its attributes: forDuration or untilDeadline. When corresponding event arrive, OnMessage and OnAlarm will activate the enclosing process task.
There are two kinds of containers of `OnMessage` and `OnAlarm`: `Pick` and `EventHandler`. The difference between `Pick` and `EventHandler` is the logical scope of effect, the explicitness, and the synchrony. The modellers will use a `Pick`, which is a specialisation of `Task` of the FlowView, in a certain part of the process control flow to specify that the process must wait for particular events to occur. The `Pick` is done when all the process tasks in the activated event are completed. The `EventHandler` is used to handle the events occurring in a certain scope -- a logic group of process tasks. If the events do not arrive, the control flow of the scope, which the `EventHandler` is attached to, is not affected or interfered by the `EventHandler`. In other words, an `EventHandler` stands loose from the rest of the process tasks in the scope such that when there are process tasks in the scope active, it is still possible to receive and handle an event specified in the `EventHandler`.

### 3.5. View manipulation mechanisms

#### 3.5.1. Extension mechanism

During the process development life cycle, various stakeholders take part in the process with different needs and responsibility. For instance, the business experts - who are familiar with business concepts and methods - sketch blueprint designs of the business process functionality using abstract and high-level and/or notational languages such as flow-charts, EPC [EPC], BPMN [BPMN], and UML Activity Diagram extensions [UML]. Based on these designs, the IT experts implement the business processes using executable languages, such as BPEL [WSBPEL]. As a consequence, these stakeholders work at different levels of abstraction of the process models.

The FlowView, CollaborationView, and the InformationView model are the cornerstones to create abstract views. These abstract views aim at representing the high-level, domain-related concepts, and therefore, they are in the first place useful the business experts. Moreover, these
views are also useful for technical experts and for enhancing the communications between different the business experts and IT experts. Nonetheless, the IT experts 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 [TZD07,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 which 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. For instance, the FlowView, CollaborationView, and InformationView model are extensions of the Core model using the generalisation relationship [TZD07,THZ+09].
Figure 18 BpelInformationView model - an extension of the InformationView model

In Figure 17, we demonstrate the extensibility of the Core model by the FlowView and the CollaborationView model as well as the extensibility of the CollaborationView model by an enhanced view model, the BpelCollaborationView model. A similar BPEL-specific extension has also been developed for the InformationView model. For the sake of readability, the extension of the InformationView model presented in Figure 13 comprises two parts that represent data processing and data structure, respectively. Figure 19(a) depicts a
BpelCollaborationView – an extension of the CollaborationView – and Figure 19(b) shows a BpelInformationView – an extension of the InformationView – of the Travel Booking process. These extension views inherit and refine the elements of their abstract counterparts.

In the same way, more specific view models for other technologies can be derived. In addition, other process concerns such as long-running transactions, event handling, human
interactions, as described above, have been extended by using new view models derived from
the basic view model using the same approach as used above.

3.5.2. Integration mechanism

In our approach, the FlowView -- as the most important concern in process-driven SOAs -- is
often used as the central view. Views can be integrated via integration points to produce a
richer view or a more thorough view of the business process. We devise a name-based
matching algorithm for realising the view integration mechanism. This algorithm is simple,
but effectively used at the view level (or model level) because from a modeller’s point of
view in reality, it makes sense, and is reasonable, to assign the same name to the modelling
entities that pose the same functionality and semantics. Nonetheless, other view integration
approaches such as those using class hierarchical structures or ontology-based structures are
applicable in our approach with reasonable effort as well [TZD07,THZ+09].

Before discussing in detail the name-based view integration, we propose the definition of
conformity of model elements and integration points. Let \( m \) be an element of a certain view
model, \( ma \) denotes the hierarchical tree of inheritance of \( m \), i.e., all elements which are
ancestors of \( m \), and \( m.x \) denotes the value of the attribute \( x \) of the element \( m \).

**Definition 1**: (Conformity) Let \( M_1, M_2 \) be two view models and \( m_1 \in M_1 \) and \( m_2 \in M_2 \). Two
elements \( m_1 \) and \( m_2 \) are conformable iff \( m_1 \) and \( m_2 \) have at least one common parent type in
their tree of inheritance or \( m_1 \) is of type \( m_2 \), or vice versa.

Using \( m_1 \uparrow m_2 \) to denote \( m_1 \) and \( m_2 \) are conformable. Definition 1 is given as:

\[
m_1 \uparrow m_2 \iff (ma_1 \cap ma_2 \neq \phi) \cup (m_1 \in ma_2) \cup (m_2 \in ma_1)
\]

**Definition 2**: (Integration point) Let \( M_1, M_2 \) be two view models and two views \( V_1, V_2 \) be
instances of \( M_1 \) and \( M_2 \), respectively. A couple of elements \( e_1 \) and \( e_2 \), where \( e_1 \in V_1 \) and \( e_2 \in
V_2 \), \( e_1 \) is an instance of \( m_1 \), and \( e_2 \) is an instance of \( m_2 \), is an integration point between \( V_1 \) and
\( V_2 \) if and only if \( m_1 \) and \( m_2 \) are conformable and \( e_1 \) and \( e_2 \) have the same value of the attribute “name”.

Using \( I(e_1, e_2) \) to denote the integration point between two views \( V_1 \) and \( V_2 \) at the elements \( e_1 \)
and \( e_2 \), and \( x \prec y \) to denote \( x \) is an instance of \( y \), Definition 2 can be written as:

\[
I(e_1, e_2 \mid e_1 \in V_1, e_2 \in V_2, e_1 \prec m_1, e_1 \prec m_2, V_1 \prec M_1, V_2 \prec M_2) \iff (m_1 \uparrow m_2) \cap (e_1 \text{name} = e_2 \text{name})
\]

The main idea of the name-based matching for view integration is to find all integration
points \( I(e_1, e_2) \) between two views \( V_1 \) and \( V_2 \) and merge these two views at those integration
points. The merging at a certain integration point \( I(e_1, e_2) \) is done by creating a new element
which aggregates the attributes and references of both \( e_1 \) and \( e_2 \). This idea is realised by the
code from line 7th to line 20th in Figure 20 [TZD07,THZ+09].
The complexity of the name-based matching algorithm is approximately $O(k + l + k.l)$, where $k = |V_1|$ and $l = |V_2|$. This complexity can be significantly reduced by generating and maintaining a configuration file containing the integration points of every pair of views with tool support. This is reasonable in reality because the integration points can be directly derived from the relationships between the FlowView and other views. For instance, a developer uses the modelling framework to realise an AtomicTask that performs a service invocation. The service invocation is defined by an Invoke activity in the BpelCollaborationView with the same name. The tooling then can automatically derive an integration point between the AtomicTask and the Invoke activity. Later on, the view integration algorithm only loads the configuration files and performs view merging rather than iterates through all elements of two input views. This way, the complexity of the view integration algorithm is reduced to approximately $O(P)$, where $P$ is the number of integration points between $V_1$ and $V_2$. We note that $P \leq k.l$. In reality, the number of elements which are

---

**Figure 20 Name-based matching algorithm**

Input: View $V_1$ and view $V_2$

Output: Integrated view $V_{12}$

```
1 begin
2  $V_{12}$.initialize()
3  $E_1 \leftarrow V_1$.getAllElements()
4  $E_2 \leftarrow V_2$.getAllElements()
5  $V_{12}$.addElement($E_1$)
6  $V_{12}$.addElement($E_2$)
7  /* look for conformable elements and integrate their attributes and
8     references into a newly created element */
9  foreach $e_i \in E_i$ do
10     $found \leftarrow$ false
11     while not $found$ do
12         $e_2 \leftarrow E_2$.next()
13         if ($e_i$.name = $e_2$.name) AND ($e_i$.superType = $e_2$.superType) then
14             $found \leftarrow$ true
15             create $e_{new}$
16             $e_{new}$.mergeAttributes($e_i$.getAttributes(), $e_2$.getAttributes())
17             $e_{new}$.mergeReferences($e_i$.getReferences(), $e_2$.getReferences())
18             $V_{12}$.addElement($e_{new}$)
19             $V_{12}$.removeElements($e_i$, $e_2$)
20     end
21  end
22 return $V_{12}$
23 end
```
used for view integration are often much less the total number of elements of the containing view, and therefore, $P << k_1l_1$ [TZD07,THZ+09].

### 3.5.3. Code generation

There are two basic types of model transformations: model-to-model and model-to-code [SV06]. 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 our view-based approach, the model transformations are merely model-to-code transformations that take as input one or many views and generate schematic code in executable process languages, such as BPEL (see Figure 21). In the literature, there are different code generation techniques such as template-based transformation, inline generation, or code weaving have been proposed [SV06]. In our proof-of-concept implementation, we use the template-based technique which has been realised using the XPand language of the openArchitectureWare framework [OAW] to implement the code generations. Nevertheless, other above-mentioned techniques can be leveraged in our approach with reasonable modifications as well [TZD07,THZ+09].

---

Figure 21 View-based model-to-code transformation (code generation)

Figure 22 shows our view-based code generator which is defined based on the view models. Existing process views can be used by the code generator to produce schematic code and runtime configurations of the process implementation. The generated schematic code and configurations sometime need to be complemented with manually written code (also called individual code) in order to fulfil particular business functionality. A typical scenario in a model-driven paradigm, according to the best practices of process-driven development, is not to embed certain special business logics in process models in order to enhance the level of abstraction, high reusability and interoperability of the models. To serve this purpose, the code generator can combine the individual code with the schematic code to produce final process implementation that can be deployed for execution in a process engine [TZD07,THZ+09].
In Figure 22(a), we present an excerpt of our template-based code generation rules in the XPand language along with the schematic BPEL code generated by using the template rules (see Figure 22(b)). Further details of the XPand language can be found in the openArchitectureWare framework documentation [OAW].

### 3.6. View-based reverse engineering

In the context of process-driven SOAs, many companies have built up an enormous repository of existing process code in executable languages, such as BPEL/WSDL, BPML, and FDL. There are three important issues that the companies are confronted with concerning the evolution of those processes. First, the legacy process code contains many tangled concerns, such as the control flow, service invocations, data processing, fault and event handling, transactions, and so forth. Second, the languages used to describe these processes are rather technology-specific, and therefore, the abstract, high-level representations are not explicitly available at the code level. Third, there is no explicit link between process design languages, such as BPMN, EPC, and UML Activity Diagram, and process implementation languages, such as BPEL and BPML. Hence, due to the mismatches of syntax and semantics and many other reasons, the cohesion of the high-level description and process code is likely obscured.

As a consequence, this hinders the adaptation of business processes to the evolution of both business environment and technology. The stakeholders, by their knowledge, experience, and skills, involve in the evolution by analysing the “as-is” processes and, if necessary, re-designing or re-developing them to adapt to new requirements, policies, technologies, etc. Unfortunately, a legacy process description, due to its complexity and technology specificity, as described in the previous paragraph, is often inappropriate to particular needs and knowledge of a certain stakeholder, especially domain/business experts. Thus, such stakeholders hardly understand, analyse, maintain, or reuse fragments of existing executable process descriptions.

Our view-based approach previously introduced can potentially resolve these issues. Using this approach, the process models are organised into separated and formalised concerns at different abstraction levels. In addition, view extension and integration mechanisms can be used to provide better view models tailored to the stakeholders’ interests. However, for budgetary reasons, developing the process views, required in our approach, from scratch is a poor and costly option. A promising alternative is an (semi-)automated re-engineering approach comprising two activities: reverse engineering for building more appropriate and
relevant representations of the legacy code and forward engineering for manipulating the process models, and for re-generating certain parts of the process code. During the reverse engineering process, high-level, abstract and low-level, technology-specific views of the process models are recovered from the existing code. In this way, the reverse engineering approach helps stakeholders to gain more appropriate representations of the processes such that they can efficiently get involved in process re-development and maintenance at different abstraction levels.

An appropriate reverse engineering of business processes, on the one hand, has to be able to adapt and tailor process models to stakeholder knowledge and needs. On the other hand, it is also able to enhance the integration, reusability, and interoperability of various process models. The view-based reverse engineering approach aims at achieving these goals. In the subsequent sections, we present this approach in detail and realise it in terms of a reverse engineering tool chain that fits into the View-based Modelling Framework.

3.6.1. View-based reverse engineering approach

![Diagram of View-based reverse engineering approach]

Figure 23 depicts the overview of the view-based reverse engineering approach that comprises a number of view-based interpreters, such as FlowView interpreter, InformationView interpreter, and CollaborationView interpreter, and so on. Each interpreter is responsible for interpreting and recovering the corresponding view from existing process descriptions. Recall that in VbMF a particular view must conform to its view model. Therefore, the interpreter of a certain view is defined based on the view model to which the corresponding view conforms. For instance, a FlowView might consist of the modelling elements such as Task, Parallel, Sequence, Exclusive, Branch, Default, and Loop according to the FlowView model described in Figure 9. In order to recover the FlowView from process descriptions, the FlowView interpreter walks through the input descriptions to retrieve related information of these elements and ignores the others [TZD08a,TZD08b].
The major objective of the view-based reverse engineering approach is to recover relevant representations of different abstraction levels from existing legacy processes. In particular, our approach extracts relevant concerns of the process represented in terms of view models by using view interpreters. The resulting views can be further integrated to adapt to specific needs, knowledge, and experience of particular stakeholders. The approach is able to produce high-level, abstract views for business analysts and technology-specific views for IT experts from existing process code base. The abstract parts extracted from the process code are integrated into VbMF using the higher-level views, whereas the technology-specific parts are integrated using the technology-specific views as described in Figure 23. Additionally, the views obtained from the reverse engineering tool chain, in turn, can be manipulated using the view integration mechanisms to produce richer views which fit better to stakeholders' interests, or to produce a more thorough view of the whole process. Any change on the views can be propagated by means of code generation provided in VbMF.

3.6.2. General approach for view extraction

The process descriptions comprise the specification of business functionality in a certain language, for instance, as we exemplify in this chapter, BPEL. Moreover, the process functionality also exposes through service interfaces, for instance, expressed in WSDL [WSDL]. To extract appropriate views from process descriptions, i.e., BPEL and WSDL specifications, we need high-level interpreters such as FlowView interpreter, CollaborationView interpreter, and InformationView interpreter as well as technology-specific interpreters such as BpelCollaborationView interpreter, BpelInformationView interpreter, and so forth.

Our general approach to define view interpreters is based on the Partial Interpreter pattern [Zdu03]. This pattern is typically applied when the relevant information to be interpreted from a language is only a (small) sub-set of the source document's language, and thus, the complexity of the whole language should be avoided in the subsequent interpretation. In particular, we concentrate on specific view models. The approach based on Partial Interpreter enables us to define modular, pluggable view interpreters, and the framework to be easily extensible with new views and view extraction interpreters. The solution is to provide a Partial Interpreter for view extraction, which only understands the specific language elements required for one view. There is a generic parser that is responsible for parsing the process descriptions. The parsing events generated by this generic parser are interpreted by the Partial Interpreters, which only interpret the language elements relevant to a particular view. Hence, the following steps are necessary for defining view extraction interpreters [TZD08a,TZD08b]:

1. Define a generic interpreter for parsing the content of the process modelling language (and other relevant languages). In the case of BPEL and WSDL, this is a generic XML parser and a parsing event model, which can be interpreted by the Partial Interpreters.
2. For each view: Define a mapping specification between the elements in the process modelling language elements and the view model elements. That is, the mapping specification contains all elements of a particular view model, and describes how they map to a sub-set of the elements in the process modelling language (and other relevant languages).
3. For each view: Define a view-specific interpretation specification that interprets only the relevant elements for a particular view from the process modelling language. That is, the Partial Interpreter specification explains how a view model can be filled with the information from the process modelling language (and other relevant languages).
The Partial Interpreter's mapping specification and view-specific interpretation specification are both defined generically on basis of the view models. Hence, they can be reused for many concrete view models.

In the subsequent sections, we present the details of the realization of the view-based reverse engineering in which abstract views such as FlowView and CollaborationView as well as technology-specific views such as BpelCollaborationView can be (semi-)automatically recovered from the existing process code.

3.6.3. Recovering abstract, high-level representations

Recovering the FlowView [TZD08a, TZD08b]

In order to recover the FlowView from BPEL code, the FlowView interpreter, which is based on the FlowView model (see Figure 24), walks through the process description in BPEL and collects the information of atomic and structured activities. Then, it creates the elements in the FlowView and assigns their attributes with relevant values as specified in Figure 24.

![Figure 24 Recovering the FlowView model from BPEL descriptions](image)

According to the specification of the FlowView model, a FlowView includes the elements representing the orchestration of process tasks. Therefore, the FlowView interpreter only considers this aspect of BPEL processes and ignores the rests.

In particular, the hierarchy and the execution order of a BPEL process are essentially defined using the control structures, such as `<sequence>` -- a step-by-step execution of activities, `<flow>` -- a concurrent execution of activities, `<if>`-`<elseif>`-`<else>` -- a conditionally exclusive branch, and `<while>` -- a condition iteration of a certain task. These control structures can be nested and combined in arbitrary manners to represent various complex control flows in BPEL processes. With respect to their syntax and semantics specified in the BPEL specification, those structures are straightforward mapped into the FlowView model.

In addition, a BPEL process might have one or many primitive activities, such as `<invoke>` -- a service invocation, `<receive>` -- waiting for a message from partners, `<reply>` -- sending back a response to a partner, and `<assign>` -- assigning values to variables and properties. Those activities are mapped into the corresponding `AtomicTask` elements of the FlowView. By abstracting these different activities away from their concrete meanings such as “receiving an XML message (i.e., an order) from a partner who plays a role of Customer (i.e., a customer)”, “invoking a service or process via the operation x of the portType p provided by the partner who plays a role of a banking institution”, etc., the resulting FlowView is merely presented to the stakeholders in form of a “plain” business logic (i.e., an orchestration of process activities) accomplishing particular business goals. In this way,
stakeholders, especially business experts, can better understand, analyse, and modify the FlowView to adapt to new requirements or changes.

**Recovering the CollaborationView [TZD08a, TZD08b]**

The CollaborationView interpreter is realised using the same approach as the FlowView interpreter. However, the CollaborationView comprises not only the elements from the process descriptions (i.e., BPEL files) but also from the service interface descriptions of the processes (i.e., WSDL files). Hence, first of all, the interpreter has to gather all information of service interfaces, messages, roles, and partners from WSDL files. Then, the interpreter creates corresponding elements and relationships in the CollaborationView according the mapping rules given in Figure 25. Next, the interpreter walks through the BPEL code to extract information from the collaborative activities. Each of the BPEL collaborative activities, such as `<invoke>`, `<receive>`, or `<reply>`, shall appear on the resulting CollaborationView with the same name as in the FlowView. However, these activities shall contain additional collaborative attributes as depicted in Figure 25.

<table>
<thead>
<tr>
<th>WSDL element</th>
<th>CollaborationView element</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;definition&gt;</code></td>
<td>cv::Service</td>
</tr>
<tr>
<td><code>&lt;message name=&quot;...&quot;&gt;</code></td>
<td>cv::Message/setName()</td>
</tr>
<tr>
<td><code>&lt;portType name=&quot;...&quot;&gt;</code></td>
<td>cv::Interface/setName()</td>
</tr>
<tr>
<td><code>&lt;operation name=&quot;...&quot;&gt;</code></td>
<td>cv::Operation/setName()</td>
</tr>
<tr>
<td><code>&lt;input name=&quot;...&quot; message=&quot;...&quot;&gt;</code></td>
<td>cv::Channel</td>
</tr>
<tr>
<td><code>&lt;output name=&quot;...&quot; message=&quot;...&quot;&gt;</code></td>
<td>cv::Channel</td>
</tr>
<tr>
<td><code>&lt;partnerLinkType name=&quot;...&quot;&gt;</code></td>
<td>cv::Partner/setName()</td>
</tr>
<tr>
<td><code>&lt;Role name=&quot;...&quot;&gt;</code></td>
<td>cv::Role/setName()</td>
</tr>
<tr>
<td><code>&lt;service name=&quot;...&quot;&gt;</code></td>
<td>cv::Service.setModuleName()</td>
</tr>
</tbody>
</table>

**Figure 25** Recovering the CollaborationView from WSDL descriptions

In addition to creating the corresponding elements of the CollaborationView, the interpreter uses the information collected in the former step to establish the necessary relationships between these elements. For instance, the relationship between `cv::Interaction` and `cv::Partner` elements is derived from the association between the communication activities (e.g., `<invoke>`, `<receive>`, `<reply>`) and the corresponding `<partnerLink>`, or the relationship between `cv::Partner` and `cv::Role` is derived from the association among the `<partnerLinkType>` and `<role>` elements in the WSDL descriptions and the `<partnerLink>` elements in the BPEL code.
3.6.4. Recovering low-level, technology-specific representations

So far the abstract, high-level representations of process models such as FlowView and CollaborationView are recovered from existing BPEL and WSDL descriptions. Out of these abstract views, the technology-specific views can be derived using the refinement mechanism with the support of IT experts. However, existing process implementations, including BPEL and WSDL descriptions, embody low-level, technical representations of the processes as well. Therefore, instead of deriving the technology-specific views from the abstract counterparts, which is error-prone and time-consuming, we leverage the view-based reverse engineering approach to parse the process implementation and recover the embodied low-level, technology-specific representations of the process. Moreover, the low-level view interpreters are developed by reusing the realisation of the abstract interpreters and enriched with additional feature according to the corresponding view models. This is likely the same manner...
that a technology-specific view inherits and enriches an abstract counterpart. This way, the integrity and consistency of an abstract view and the corresponding technology-specific views are maintained because they are both recovered from the same inputs. Nonetheless, maintaining the consistency requires additional supports by the tooling and is one of our future works.

We illustrate the recovering of low-level, technology-specific views using the view-based reverse engineering approach in Figure 26. The CollaborationView is a high-level representation of the BpelCollaborationView, which is at a lower level of abstraction. The BpelCollaborationView enriches the CollaborationView with either newly added elements or inherited elements with additional properties. In Figure 26, the right column shows the elements of the BpelCollaborationView recovered from the BPEL descriptions. Many of those elements are partially inherited the information of the relevant elements of the CollaborationView shown in the middle column of Figure 26. Thus, we can inherit the corresponding parsing elements from the CollaborationView interpreter and only develop parsers for additional features of the BpelCollaborationView.

3.7. Tool support

3.7.1. View-based, model-driven forward engineering

The view models, components and model manipulation mechanisms mentioned above are the essential parts shaping the view-based, model-driven development tool chain shown in Figure 27. The arrows conceptually depict the flow of data. This tool chain supports stakeholders in modelling and implementing business process in a forward engineering approach.

![Figure 27 View-based, model-driven development tool chain](image-url)
In this tool chain, business experts, who are familiar with domain- and business-oriented concepts and knowledge, involve in process development by using the VbMF abstract views. The IT experts, who mostly work with technical details to implement and deploy business processes, refine the abstract views into technology-specific views. During the course of modelling or implementation, stakeholders, either business experts or IT experts, can utilise the view integration mechanism in order to achieve an appropriate view or a more thorough view of the process. After the stakeholders finish developing the corresponding VbMF views, the code generators can be used to produce schematic process code in terms of BPEL and WSDL as well as necessary process deployment configurations. The IT experts may involve in developing individual code (or so-called manually written code) that complements the generated schematic code with specific business logics or deployment configurations. The aforementioned development process can be iterated until process functionality is fully accomplished to fulfil a certain business goal.

We realized the aforementioned tool chain by facilitating the Eclipse Modelling Framework (EMF) and openArchitectureWare MDD framework (see Figure 28). The greatest advantage of using the Eclipse Modelling Framework is that we gain a better integration and interoperability with existing development tools developed based on EMF Ecore, a MOF-compliant meta-model, and XMI -- a standard for serializing models. The View-based Modelling Framework provides different editors for manipulating views, such as the FlowView Editor, the (Bpel)CollaborationView Editor, the (Bpel)InformationView Editor, and so on. For the sake of demonstration, we mainly use the tree-based editors which are generated from our view models by the EMF generator. The template-based code generation rules are developed using the XPand language editor provided in openArchitectureWare framework. Using these rules, we can automatically generate process implementations including BPEL and WSDL descriptions.

Many tasks of the modelling tool chain can be automated through the workflow engine provided by openArchitectureWare MDD framework. The openArchitectureWare workflow engine takes a declarative XML-based configuration, namely, a workflow. A workflow is typically made up of various workflow components which are POJOs derived from oAW predefined interfaces. These workflow components can be used for reading and instantiating models, checking them for OCL-based constraint violations, transforming models, and the finally, for generating code. VbMF provided a predefined workflow for loading process views, checking the conformity of views against the corresponding view models, and generating process implementations in BPEL and WSDL as well as process deployment configurations for some BPEL engines such as ActiveBPEL [ActiveBPEL] and Apache ODE [ODE] (see Listing 1).
Figure 28 The proof-of-concept tooling of the view-based, model-driven approach

```xml
<workflow>
  <!-- Declare VbMF view models -->
  <bean class="org.eclipse.mwe.emf.StandaloneSetup">
    <registerEcoreFile value="framework/model/vb/core.ecore"/>
    <registerEcoreFile value="framework/model/vb/flow.ecore"/>
    <registerEcoreFile value="framework/model/vb/collaboration.ecore"/>
    <registerEcoreFile value="framework/model/vb/information.ecore"/>
    <registerEcoreFile value="framework/model/vb/bpelcollaboration.ecore"/>
    <registerEcoreFile value="framework/model/vb/bpelinformation.ecore"/>
  </bean>

  <component class="org.eclipse.mwe.emf.Reader">
    <uri value="$core"/>
    <modelSlot value="core"/>
  </component>

  <component class="org.eclipse.mwe.emf.Reader">
    <uri value="$flow"/>
  </component>
</workflow>
```
<modelSlot value="fv"/>
</component>

<component class="org.eclipse.mwe.emf.Reader">
  <uri value="${information}" />
  <modelSlot value="iv" />
</component>

<component class="org.eclipse.mwe.emf.Reader">
  <uri value="${collaboration}" />
  <modelSlot value="cv" />
</component>

<component class="org.eclipse.mwe.emf.Reader">
  <uri value="${event}" />
  <modelSlot value="ev" />
</component>

<!-- Generate WSBPEL 2.0 process implementation -->
<component class="org.openarchitectureware.xpand2.Generator">
  <metaModel idRef="vbmm" />
  <expand value="framework::template::wsbpel::WSBPEL({fv, iv, cv, ev}) FOR core" />
</component>

<!-- Generate ActiveBPEL deployment configuration -->
<component class="org.openarchitectureware.xpand2.Generator">
  <metaModel idRef="vbmm" />
  <expand value="framework::template::pdd::PDD({fv, iv, cv, ev}) FOR core" />
</component>

<component class="org.openarchitectureware.xpand2.Generator">
  <metaModel idRef="vbmm" />
  <expand value="framework::template::wsdlCatalog::wsdlCatalog({fv, iv, cv, ev}) FOR core" />
</component>

<!-- Generate Apache ODE configuration -->
<component class="org.openarchitectureware.xpand2.Generator">
  <metaModel idRef="vbmm" />
  <expand value="framework::template::odeDD::DD({fv, iv, cv, ev}) FOR core" />
</component>
</workflow>

Listing 1 A workflow for generating process code and deployment configurations
3.7.2. View-based reverse engineering

The view-based reverse engineering approach described in the previous sections has been realized to form a tool chain that supports stakeholders in extracting appropriate representations of process descriptions in terms of high-level or low-level, technology-specific view models (see Figure 29).

![View-based reverse engineering tool chain for process-driven SOAs](image)

**Figure 29** View-based reverse engineering tool chain for process-driven SOAs

In this reverse engineering tool chain, existing process implementations in terms of BPEL and WSDL descriptions, and process deployment configurations in XML-based representations, are taken as inputs for the various view-based interpreters, respectively. The outcomes of these view-based interpreters are either VbMF abstract views -- the high-level representations of processes or technology-specific views -- the low-level representations of processes. The stakeholders, according to their particular needs, knowledge, and experience, can work with their representations of choice. For instance, business experts will mainly involve in the abstract, high-level process views whilst IT experts are more interested in the technology-specific views.

We have developed the view-based interpreters using XPath technology [XPath] to extract XML-based information from process implementations such as BPEL and WSDL and build up corresponding in-memory views. After that, these in-memory views are then serialised and stored in model repositories in terms of XMI-compliant [XMI] descriptions by using model serialisation techniques provided in the Eclipse Modelling Framework. These process views, in terms of XMI-compliant descriptions, can be (re-)used later in VbMF.
3.8. Summary

So far we have presented our view-based, model-driven approach for process-driven SOAs. We elaborated on how the notion of views have been exploited for dealing with the complexity of the horizontal dimension, i.e., the dimension of different concerns, of process-driven SOA development, and how the model-driven development paradigm is leveraged for the separation of abstraction levels. We illustrated the realisation of these concepts in terms of a View-based Modelling Framework for process-driven SOA development. In this framework, the notion of views is central. We developed a number of view models for formalising process concerns, such as the control flow, service interactions, and data processing as well as additional concerns, such as human interactions and transaction. In order to provide view models which 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 such that business experts can easily understand and manipulate them to accomplish a certain business goal. The technology-specific layer consists of view models that are merely relevant to IT experts who are responsible for implementing, deploying, and maintaining the processes. This combination of the separation of concerns principle and the separation of abstraction levels offers a flexible, extensible methodology for process-driven SOA development.

Furthermore, our framework also offers different modelling and development mechanisms to stakeholders. The various editors that embody the view extension and view integration mechanisms support stakeholders in creating and manipulating view models and views. In addition, the code generator generates process implementations from views, and therefore, enhances the automation and productivity in process development. These mechanisms, along with the aforementioned concepts, shape out forward engineering tool chain for process-driven SOAs.

The view-based reverse engineering approach can help the various stakeholders of a process-driven SOA to overcome two important issues. Firstly, it exploits the notion of views to deal with the complexity of existing process repositories and to adapt the process representations to the stakeholders' interests. Secondly, it provides the ability of integrating diverse process models and offers explicit relationships for understanding and maintaining process models and for propagating changes. Hence, process models at different abstraction levels and different process concerns can be reused for developing the others. This has been achieved by developing a novel concept for a reverse engineering tool chain, based on partial interpreters and view models, and by seamlessly integrating this reverse engineering tool chain into our View-based Modelling Framework, which also supports means for forward engineering, such as view integration, view extension, and code generation. The view-based reverse engineering approach enables the reuse of existing process code, e.g. implemented in BPEL and WSDL, in the View-based Modelling Framework.

4. Domain-specific languages in COMPAS

Domain-specific languages (DSLs) are languages that are specifically tailored for the needs of a particular problem or application domain. They promise that domain experts themselves can understand, validate, modify, test, and sometimes even develop DSL programs. DSLs can help as they come with the promise to enable involving domain experts more closely in the software design activities (see [D1.1]).
In general, DSLs can be designed and used on different abstraction layers, ranging from DSLs for technical tasks to DSLs for tasks on the business-level. Thus, DSLs can be defined for non-technical stakeholders, such as business analysts or biologists, for example. In general, a DSL makes domain-knowledge explicit. That is, the DSL is built so that domain experts can understand and modify DSL code to phrase domain-specific statements that are understood by an information system. An important design principle of DSLs is therefore to keep the DSL as simple as possible while making it as powerful as needed.

In the following, we first describe the major architectural decision and design process aspects, relevant for the COMPAS DSLs – as a foundation for DSL design in COMPAS. At the end of this section, we describe how to build DSLs for compliance concerns, and introduce the COMPAS QoS DSL as an example.

Making architectural decisions in a particular DSL development project is not always easy because many of the decisions that must be taken by the architect may have important consequences for other parts of the project [ZS09].

The typical sequence of the three decisions is to first determine the DSL development process (Section 4.1), second choose the concrete syntax style (Section 4.2), and third decide whether implementing an external vs. embedded DSL best suits the needs of a particular DSL project (Section 4.3).

Figure 30 shows the main relations between the decisions. Relationships between decisions are mainly mandatory or optional “follow-on decision” relationships. Most often a follow-on decision is only selected if particular alternatives are selected or applied in specific circumstances. Hence, we use the rather vague “can influence the decision” relationships.

4.1. DSL development process

DSL development is usually embedded in some general purpose software development process, such as the Unified Process or an agile process. In addition, there are many DSL project-specific factors that require special attention, such as end-user or domain expert involvement. These factors lead to the need to choose a development micro-process that best fits a particular DSL project [ZS09].
Figure 31 Decision overview: DSL development process

Figure 31 illustrates the alternative solutions for the architectural decision of the DSL development process.

4.1.1. Language model driven DSL development

In this type of DSL engineering process, the language model definition drives the DSL development. That is, first the core language model is defined to reflect all relevant domain abstractions, then the concrete syntax is defined along with the DSL’s behaviour, and finally the DSL is mapped to the platform/infrastructure on which the DSL runs. Like all other process variants, the language model driven process is working incrementally. This means, incremental refinements, iterations, and feedback loops are applied whenever necessary, for instance to correct mistakes or to refine the results of prior activities. The language model driven process variant can be further tailored by specifying how large the increment defined in an iteration is – ranging from larger language model chunks that are defined in one batch, to an own iteration of the process for every single language model element [SZ09].
Figure 32 shows an example tailoring of the DSL engineering process. In particular, the process in Figure 32 shows the main control and object flow of the “Language Model Driven Process” variant. Note that it is of course possible to move back in the process at any time, for example, if an error or incompleteness in one of the models is detected. For the sake of simplicity, however, we chose to omit the corresponding reverse links in Figure 32 and focus on the main flows. The purpose of this and the following figures in this section is two-fold:

- First, the figures specify each action (or sub-activity) in terms of the inputs required and the outputs provided by this particular action (or sub-activity).
- Second, the figures show one possible tailoring of the process as a control and object flow. The purpose is to show to the reader one possible way of how to combine the DSL engineering activities and traverse through a DSL process. In particular, we use the “Language Model Driven Process” for explanatory purposes.

In most cases you should conduct the two activities of defining the DSL’s behaviour and the
concrete syntax in parallel because the behaviour and the concrete syntax of a DSL may mutually affect each other. Performing the two activities in parallel is especially useful for embedded DSLs because syntax and behaviour of the host language have a considerable influence on the concrete syntax and behaviour of the DSL. Moreover, conducting both activities in parallel is often useful because intermediate results can be exchanged to produce a concrete syntax and corresponding behaviour definitions, which are consistent, complete, and well integrated with each other.

After defining the DSL’s behaviour and concrete syntax, we need to map the different artefacts to the target platform. This activity produces transformations and corresponding integration tests for the DSL. Furthermore, if the target platform needs to get extended or adapted in the course of the DSL development, we need to produce the corresponding platform extensions [SZ09].

4.1.2. Mockup-language driven DSL development

In case a domain is very complex and cannot be easily understood by the software engineers (e.g., air traffic control, definition of financial products, or power-plant control) DSL engineering requires the domain experts to be highly available for the development team. In such a project, it is sensible to raise the participation of the domain experts through the initial definition of a mockup language [SZ09].

In particular, we start DSL development with the concrete syntax design and then distil the abstract syntax and semantics from this concrete syntax, where the concrete syntax is developed together with a domain expert by utilizing the domain expert’s domain knowledge. Next, a first prototype implementation of the resulting mockup language is developed. The prototype then serves as a means to perform acceptance-testing activities, to further refine and tailor the concrete syntax to the domain expert’s needs. This is done in multiple iterations to get closer to the domain abstractions in a stepwise manner [SZ09].

In this process, the initial DSL prototype serves as a means to perform acceptance-testing activities, to further refine and tailor the concrete syntax to the domain expert’s needs. This is done in multiple iterations to get closer to the domain abstractions in a stepwise manner [ZS09].

That means, the sub process for definition of the concrete syntax is guiding the whole DSL development process. In particular, definition of the concrete syntax can either be performed iteratively (per language element) or a version of the concrete syntax can be completely defined (including all perceived language elements). For instance, in case a second concrete syntax is added to a DSL, it sometimes makes sense to specify the whole syntax at once. In our experience, a more iterative approach makes sense in case the language is used for illustrating and sketching domain concepts incrementally while the DSL is developed. This, of course, is dependent on the availability of the domain experts. Often the mockup language driven DSL development process is well suited for projects that follow an agile development approach, which advocates onsite domain experts (customers) [ZS09].

4.1.3. Extracting the DSL from an existing system

Sometimes an existing software system should (ex post) be provided with a DSL. That is, the domain abstractions for the DSL can be derived directly from the existing system. In this case, it makes sense to first elicit the language model elements from the abstractions given in that system. If the architecture of the corresponding software system is documented (e.g., using a graphical modelling language such as the UML), this architecture description can be a
valuable source for the elicitation of domain abstractions and DSL behaviour. Here, domain experts either take part in the elicitation process, or they review the resulting language model. The process can be further tailored depending on the availability of the domain experts and on how iterative the development process is [ZS09, SZ09].

4.2. Concrete syntax style

The development of the concrete syntax for a DSL includes the supporting frontend components, such as parsers or graphical editors. Figure 33 illustrates the alternative solutions for this architectural decision.

4.2.1. Textual concrete syntax

A textual concrete syntax should be used if users of the DSLs primarily use text-based input to develop models in the DSL. The textual concrete Syntax is processed using a parser. The parser can be developed from scratch using a parser generator, such as Antlr\(^1\) or COCO/R\(^2\). Alternatively, the interpretation infrastructure of an existing (programming) language can be customized to enable DSL parsing. Often scripting languages and dynamic languages, such as Ruby\(^3\), Tcl\(^4\), Python\(^5\), or Perl\(^6\) are used in this context, as they are easy to extend with new language elements.

In both cases, a mapping to the abstract syntax of the DSL is needed. That is, if DSL text is parsed, the output of the parsing step must provide a model conforming to the DSL’s abstract syntax.

In any case, a textual concrete syntax can be edited with an ordinary text editor. However,

\(^1\) http://www.antlr.org/
\(^2\) http://www.ssw.uni-linz.ac.at/coco/
\(^3\) http://ruby-lang.org/
\(^4\) http://tcl.sourceforge.net/
\(^5\) http://www.python.org/
\(^6\) http://www.perl.org/
often it makes sense to provide custom editors for the DSL including features such as error highlighting, syntax highlighting, help texts, and so on. In case a custom parser has been written, this step is mandatory if more support than simple text editing is needed. If an interpreted (programming) language is customized (i.e., extended with DSL-specific language abstractions), editors for the corresponding host language can be used to phrase DSL expressions – even though such editors most often do not provide native support for the DSL elements. If the customization is close to the original syntax of the host language, it is often possible to modify the existing editors. Otherwise a custom text editor is needed as well [ZS09].

4.2.2. Textual concrete syntax with generation of visualisations

In order to visualise DSL programs or models, it is often not too much effort to provide a graphical concrete syntax editor. If the graphical concrete syntax editor is not necessary, the DSL models can be rendered either for printing or presentation in a GUI, e.g., via tools like Graphviz\(^7\) or Prefuse\(^8\).

4.2.3. Graphical concrete syntax

In graphical concrete syntaxes, often the resulting DSL models/programs are saved or exported by the editor tool in an interchange format, such as XMI. The interchange format is then parsed by another tool, which maps the model to the abstract syntax of the DSL. In integrated tool suites, containing both a graphical editor and a generator in one tool, another option is to directly transform the internal memory representation of the model used inside the graphical editor to the abstract syntax of the DSL – i.e. without going through an interchange format [SZ09].

That is, the development of a graphical editor and a mapping from the internal representation used by this editor to the abstract syntax of the DSL is needed. In addition, depending on the use cases of the DSL (resp. of DSL models/programs) optional export and import components for an interchange format such as XMI are required.

4.2.4. Form- or Table-based Concrete Syntax

Usually form- or table-based concrete syntaxes come with a respective form or table editor. In some cases only a purely textual input syntax is offered to DSL users (e.g., something akin to a comma separated values (CSV) format). The form or table is then saved in this format, and later parsed and mapped to the abstract syntax. Sometimes such textual syntaxes are also used as interchange formats.

That is, if one chooses this type of concrete syntax, in most cases the development of a tailored form or table editor is needed. A mapping from the internal (i.e., in-memory) representation for working with the form or table data to the abstract syntax is always needed. In addition, depending on the use cases of the DSL optional export and import components for an interchange format are required [SZ09].

\(^7\) http://www.graphviz.org/

\(^8\) http://prefuse.org/
4.3. External vs. embedded DSL

Figure 34 Decision overview: External vs. Embedded DSL

Figure 34 illustrates the alternative solutions for this architectural decision. In the following, the different alternatives for realizing a concrete syntax and mapping it to an abstract syntax are explained.

4.3.1. External DSL

An external DSL is defined in a different format than the intended target language(s)/platform(s) and can use all kinds of syntactical elements (independent of any other language) [ZS09].

Fowler [Fow09] presents a number of techniques for implementing external DSLs in pattern form, such as tree construction for creating a syntax tree from source code using a parser or embedded translation for embedding interpretation code into the grammar.

4.3.2. Embedded DSL

An embedded DSL (also called internal DSL) is defined as an extension to an existing programming language and uses the syntactic elements of the underlying (host) language [ZS09].

Fowler [Fow09] presents a number of techniques for implementing embedded DSLs (he uses the term “internal DSL”) in pattern form, such as nested closure used to express statement sub-elements of a function call by putting them into a closure in an argument or literal extension used to add methods to program literals.
Figure 35 Our incremental DSL development approach

Figure 35 illustrates an excerpt of our incremental DSL development process to demonstrate how to start developing DSLs to provide in a rapid manner a prototype to the DSL’s stakeholders.

Following the typical DSL development process, reported often in the literature ([KT08], [Fow09], [MHS05]), our approach usually starts with the definition of the domain models. However, it is also possible to use it with other DSL processes (see [SZ09]). The best way for language developers for gathering domain knowledge is to collaborate with the domain experts. Furthermore, the domain experts can express the requirements for the DSL straight to the language developers. To avoid misunderstandings of the domain concepts, the phase of building the domain model should be performed together by the domain experts and the language developers too.

In our approach, the definition of the domain model equates the implementation of the internal DSL, which represents the prototype of the DSL. That is, after the definition of the domain model, the DSL’s stakeholders can give feedback if all requirements were met, and if the prototype of the DSL language represents the concepts of the domain and their relations correctly.

If the stakeholders are satisfied with the prototype, the language developers can start to develop an external syntax, if it is needed. This means mainly developing a parser or graphical editor for the external syntax. To provide a running system of the system specified in the DSL, the implementation of a code generator or interpreter is needed. In our approach, the code generators or interpreters are not implemented before the first DSL prototype.

Please note that the same incremental development approach can be used for each feature of
the DSL. That is, if for some features an external syntax and code generators or interpreters have already been created, this does not hinder the incremental evolution of other features, again only using the internal syntax based on the domain model for rapid prototyping and evolution.

All components of the external DSL, code generators, and interpreters depend on the domain model. During their development, it is possible that the stakeholders update their requirements for the DSL. Hence, the language developers have to sit together with the domain experts again to discuss the updated requirements. This results in changes in the domain model, in the external syntax, in the parser of the external syntax, and in the code generators or interpreters. Again, in our approach, the language developers have to collaborate with the domain experts until the internal DSL - a prototype of the DSL - is finished. Afterward, the changes can be taken into the updating process of the external DSL and the code generators or interpreters. Hence, even if additional components have been developed for a DSL, we can follow an incremental evolution approach.

In the following Section, a language workbench is presented which can be used to provide – in an easy and rapid way – the DSL’s prototypes to its foreseen stakeholders. First, the language workbench is explained in more detail. Next, the language workbench’s features are exemplified for the development life cycle of the DSL for specifying QoS concerns.

4.4. DSLs for compliance concerns

To offer expressive and convenient languages for the different stakeholders, our approach provides a separation of DSLs into multiple sub-languages, where each sub-language is tailored for the appropriate stakeholders – see [OZD09] for further details. Hence, it is possible to provide multiple different levels of abstractions where each level of abstraction is tailored for the designated stakeholders. The number of different levels of abstractions depends on the problem domain, as well as on the number of the different stakeholders. Hence, we can better provide tailored and user-friendly representations for each group of the stakeholders.

The following sections exemplify our approach of providing high-level and low-level DSLs by an example. The example illustrates (1) how compliance concerns – especially QoS compliance concerns – can be integrated into VbMF, (2) how the low-level DSL extends the high-level DSL with the additionally needed technical aspects for expressing QoS compliance concerns of business processes, and (3) how domain and technical experts can use the high-level and low-level DSLs, respectively.

During the development of the illustrated QoS DSL, we decided to follow the language model driven DSL development approach because the QoS domain is narrow and the requirements on the DSL were not specified exactly. Because of the foreseen changing requirements, we knew that the language model will evolve permanently, and hence, we decided to implement an embedded DSL first, which presents the offered elements to the user and was used as a prototype of the final DSL. After a quite stable version of the language model, which consists of elements of the QoS domain, the code generator was implemented. Afterwards, we decided to offer an external high-level DSL with a textual syntax to the QoS experts. The implementation of a parser and a mapping, which maps the externally defined QoS compliance concerns onto the language model, was necessary.
4.4.1. The language models

4.4.1.i. The High-Level QoS DSL

The high-level QoS DSL has the requirement that domain experts should be able to model which QoS compliance concerns have to be measured for a certain business process to fulfill some Service-Level Agreements (SLAs), as well as the actions to be taken when the SLAs are violated. The high-level DSL should provide expressive notations for representing concepts and terminologies of the QoS and SLA domains.

The scope of the high-level QoS DSL covers the annotation of services and processes with QoS measurements. Each QoS measurement is defined in an SLA between the service provider and the service consumer. In this work we concentrate on the specification of runtime and performance related QoS measurements as specified in [Ran04]. An example of a high-level specification of the QoS compliance concerns is: "The availability of the service X must be greater than 99%".

![Diagram](image_url)

**Figure 36 An example of annotating processes or services with QoS compliance concerns**

Figure 36 illustrates how services of the VbMF’s core model can be annotated with QoS compliance concerns. Each service that becomes annotated with QoS compliance concerns must be of the type QoSService. SLAs contain all services and their QoS concerns. In the COMPAS project, the following four QoS compliance concerns are needed: Availability, Delivery Rate, Processing Time, and Frame Rate. For their detailed description, please refer to [D5.4]. Furthermore, a possibility for specifying Events, which can be fired during the runtime, is provided.

4.4.1.ii. The Low-Level QoS DSL

The language model of the low-level DSL extends the high-level language model for modeling the technological aspects to generate a running system from the model instances described by using the DSLs. Similar to the high-level DSLs, the provided constructs and expressions of the low-level DSL are named similar to the appropriate technology to enable technical experts to easily make the link to the technology. The expressions of the low-level QoS DSL depend on the technology on which the DSL is based.

Technical experts need a language for specifying how the different QoS values are measured in the used Web service framework, as well as how the defined actions are executed or
performed. We decided to use the open-source Apache CXF Web service framework\(^9\) in our prototype.

The technical requirements on the low-level QoS DSL are as follows: The message-flows between the service client and the service provider are based on chains. Each service client and service provider has two chains. An incoming chain is responsible for incoming messages, and an outgoing chain is responsible for outgoing messages. Each chain - incoming and outgoing - consists of a number of phases, in which the QoS values can be measured. Every phase can contain one or more interceptors, which are implemented in Java and are responsible for measuring the QoS values or for performing the required actions. In this case, interceptors do not need to be specified within the models because just by specifying the QoS measurements and the corresponding phases, the interceptors can be generated automatically.

Figure 37 illustrates the extension of the high-level QoS DSL model with the low-level ones, which provide the facilities to specify the needed technical aspects to be able to generate a running system. As already mentioned, each QoS concern of the Apache CXF web service framework (described by the `CXFQoS` class) has to be measured in one `Chain` and one or two corresponding `Phases`.

![Figure 37](image-url)

**Figure 37** Extending the high-level QoS language model with the additionally needed technical details

After the modeling of the language models, the implementation of them can take place. Nowadays many frameworks for a simple implementation of the language model exist, such

\(^9\) [http://cxf.apache.org](http://cxf.apache.org)
as EMF or the MS DSL Tools. The following Section describes how the language models of the QoS DSL are implemented by using Frag.\(^\text{10}\)

### 4.4.2. Implementation

Frag is a dynamic programming language implemented in Java. Its main goal is the rapid definition of domain-specific languages (DSLs) based on Frag and processing of these languages. DSL development in Frag is based on an explicit language model. This language model is defined in a framework called FMF. Automatically, after the definition of the language model, a concrete syntax of the DSL is defined, this is identical to the Frag syntax used for the DSL definition. In the following, the implementation of the QoS DSL’s language models is described in more detail.

#### 4.4.2.i. The High-Level QoS DSL

![Diagram of the high-level domain model using Frag](image)

Figure 38 shows how models (in UML2 notation) can be implemented in Frag. First, the class `QoSService` is created, which has the `Service` class of the VbMF’s core model as superclass (see Figure 7). Second, the class `SLA` is defined. After the definition of both classes, a

10 [http://frag.sourceforge.net](http://frag.sourceforge.net)
relationship between them can be defined. In our case, a Composition is defined, which defines that an SLA consists of multiple Services. Similar, the relation between Services and its QoS compliance concerns are defined.

After the definition of the high-level language model, the low-level language model can be implemented. The following section gives some closer insights.

4.4.2.ii. The Low-Level QoS DSL

Figure 39 shows that the low-level language model is implemented similar to the implementation of the high-level language model. First, the classes of the language model are defined and then their relationships. First the classes CxfQoS, Chain, and Phase are defined, and then their relationships, such as that Chains consist of many phases (expressed by the ChainPhases relationship), and that the QoS values have to be measured in one or two phases of a certain chain (expressed by the QoSPhase and QoSChain relationships).

4.4.3. Using the DSLs

In this section the usage of the high- and low-level DSLs for specifying the QoS compliance concerns is demonstrated. First, the high-level DSL is presented, and then, the low-level one.
4.4.3.i. Using the High-Level QoS DSL

Figure 40 shows the implemented editor for annotating services with QoS compliance concerns. Hence, the high-level QoS DSL has an external textual syntax. This means that the implementation of a parser was necessary, as well as a mapping, which maps the parsed elements onto the language model of the high-level QoS DSL.

After loading an instance of the VbMF’s core model, all services that are provided by the loaded business process, are listed in the text area. Now, the services can be annotated with QoS compliance concerns. As shown in Figure 40, the process provides the Search and Buy services where both get annotated with QoS compliance concerns. The Search service must have an availability of greater-than 99%, as well as a maximum processing time of 5 seconds. Similar, the Buy service must only have an availability of 99%.

For the time being, the editor provides the facility for checking the validity of the defined QoS annotations, such as if the concrete syntax is correct. Because the user can add any services to the list, a name-based matching is performed, to check if all specified services are in the model instance of the VbMF’s core model too. If some listed services are not in the core model’s instance, the QoS annotations are not valid.

After the specification of all QoS requirements, the high-level QoS concerns have to be extended with some technical aspects to be able to generate a running system. The following Section provides an example of using the low-level DSL.

4.4.3.ii. Using the Low-Level QoS DSL

Listing 2 illustrates an excerpt of the low-level QoS DSL, which is implemented as an embedded DSL, having a textual concrete syntax. This means that the low-level QoS DSL is implemented in Frag and offers to the end-user of the DSL the Frag syntax.

In our work, services should have an URI and a package name. This attributes can be set as default too, making it possible that the technical aspects just have to be defined once and can be used for multiple high-level descriptions.
Now, the advantages of our separation into high- and low-level languages can be seen. The QoS values have to be measured always in the same phases of the Apache CXF Web service framework. Hence, the technical experts have to specify the technological aspects just once. The requirements of the used technology do not change as often as the compliance concerns of process-driven SOAs. That is, the low-level requirements do not change as often as the high-level ones. In case the underlying technology changes, for instance, due to a software update, it is likely that the technical aspects have to be re-modelled to accommodate these changes. By extending the number of QoS measurements, such as by adding scalability and throughput measurements, the main work lies in the extension of the code generator which generates executable code.

After specifying the high- and low-level concerns of the service-oriented system, the code can be generated. In our case, the Apache CXF web services and interceptors are generated for measuring the QoS requirements on the services. The following Section provides some more insights into the developed code generator for the QoS requirements.

### 4.4.4. Generated code

In this section, some insights are given about the generated code, which are Apache CXF interceptors for measuring the QoS requirements. The interceptors are performed when the services are called. Furthermore, a service host is generated that hosts all services which were defined by using the DSLs (see Figure 40 and Listing 2). In our case, the service host hosts the *Search* and *Buy* services. Because the *Search* service must have a certain processing time (see Figure 40), and the processing time is measured on the server-side, Listing 3 illustrated excerpt of a generated interceptor is always executed when the *Search* service is called.

```java
public class ProcessingTimeInterceptor extends AbstractPhaseInterceptor<Message> {
    public ProcessingTimeInterceptor(String sPhase) {
        super(sPhase);
    }

    public void handleFault(Message msg) {
        /* do some error handling */
    }

    public void handleMessage(Message msg) throws Fault {
        if((this.getPhase().equalsIgnoreCase(Phase.INVOKE))) {
            QoSContent qos = (QoSContent)msg.get(QoSContent.class);
            if(qos==null) {
                qos = new QoSContent();
            }
            qos.setProcessingTime(System.currentTimeMillis);
            msg.setContent(QoSContent.class, qos);
        } else if((this.getPhase().equalsIgnoreCase(Phase.POST_INVOKE))){
            QoSContent qos=(QoSContent)msg.getContent(QoSContent.class);
            if(qos!=null) {
                long nRespTime = System.currentTimeMillis()-qos.getProcessingTime();
                System.out.println("ProcessingTimeInterceptor => "+nRespTime);
                qos.setProcessingTime(nRespTime);
            } else {
                throw new Fault(new Exception("no response time data found in header!"));
            }
        }
    }
}
```

**Listing 3** A generated interceptor for measuring the processing time of a service

An Apache CXF interceptor has a constructor, telling in which phase the interceptor will be executed, a `handleFault` method, for handling occurred errors, and a `handleMessage` method, which contains the logic of the interceptor. In our example, the processing time of a service has to be measured between the *INVoke* and *POST-INVoke* phases of the Apache CXF
framework. Hence, in the *INVOKE* phase, the current time is measured and placed into the content of the message (by calling the `setContent` method of the `Message` object). In the POST-INVOKE phase the previous measured time is taken, and the difference to the current time is calculated which provides the time for processing the service.

For the time being, the measured timestamps are only printed to the `System.out` handle. Hence, runtime QoS values [Ran04], such as the availability, cannot be measured. In comparison to the processing time, which can be measured at every service request, the availability of a service must be measured and calculated permanently during the whole lifetime of the service. Currently we are implementing a monitor component which can be accessed via a Web service and which stores the measured QoS values into a database. In this case, within each service a thread can be generated which sends periodically *Alive*-messages to the QoS monitor. Hence, the availability of every service can be calculated based on the number of received *Alive*-messages in comparison to the time range. Furthermore, we want to provide a possibility for a virtualization of the measured QoS values for each service, such as a web page.

### 4.5. Summary

Developing DSLs brings many design decisions. Mainly on deciding on an external vs. embedded DSL, textual vs. graphical concrete syntax, or design decisions on the development process itself. In this section, hints on choosing the right design decisions are given, based on gathered experiences in research and industrial case studies. Furthermore, our incremental DSL development process for developing rapid prototypes of DSLs, which can be demonstrated to the DSL’s stakeholders, is explained.

The current state of the development process of the QoS DSL within the COMPAS project is presented based on the described experiences on designing and implementing DSLs. The QoS DSL is separated into two sub-languages: (1) a high-level language, which is developed for business experts, providing an external syntax for annotating the services with QoS compliance concerns without technical details, and (2) a low-level language, tailored for technical experts to extend the high-level requirements with technical ones to be able to generate a running system. Both, high- and low-level languages, are developed in Frag.

Future work on the QoS DSL will include the possibility of specifying ranges, such as *min* or *max* values of certain QoS measurements. The specification of QoS policies and their interdependencies should be possible. Furthermore, it should be possible to define date and time ranges in which the specified SLAs should be satisfied, such as “*The Service X should have an Availability of 99% between Monday and Friday from 7am to 6pm*”. The low-level DSL can be extended with the specification of the data types of the offered QoS measurements, such as using `Boolean` for monitoring the availability and using `Float` for the calculated availability over a specific time range.

### 5. Prototype Integration

#### 5.1. Introduction

There are many different kinds of business compliance that companies have to consider, for instance, jurisdictional policies, business policies, QoS policies, intellectual property and licenses, and security policies [DoW]. Each of those compliance concerns embodies distinct concepts and constraints which are merely interpreted by domain experts or compliance
specialists. Therefore, the DSLs presented so far can support those experts in better eliciting such compliance concepts and constraints and applying the resulting compliance concerns in the specific context of business processes.

In Figure 41, we illustrate the modelling of process-driven systems and the compliance concerns exemplified using MDSD Software Framework. On the left hand side, the modelling framework provides fundamental view models for describing the functionality of software and systems in terms of business processes. As stated in previous sections, three view models, the Flow, Collaboration, and Information view models, represent the basic concerns of a business process. Other concerns, such as transactions, human integration, event handling, etc., are also developed and plugged into VbMF accordingly thanks to its extensibility. For the sake of simplicity and concentration on compliance modelling, these other view models are not presented in Figure 41.

On the right-hand side, VbMF will provide capabilities for specifying compliance concerns in terms of appropriate DSLs, such as DSLs for representing security policy, QoS policy, intellectual property and licenses, and regulatory or legislative provisions. Each DSL is correspondent to an extension view model. That is, these DSLs represent extensional view models for expressing compliance controls in parts of the SOA that are not directly related to processes and services. Using extension mechanisms described in Section 3.5.1, the framework can be extended to other kinds of compliance concerns. More details on extending the framework with additional compliance DSLs are elaborated in the subsequent sections.

Whereas the DSLs mentioned above define compliance controls in specific areas, namely, security, QoS, intellectual property and licenses, and regulatory or legislative provisions, there is another DSL that has a special purpose: To annotate the controls defined both in the basic VbMF views as well as the controls defined in the four extensional DSLs with compliance metadata. Using the Compliance Metadata model all compliance controls can be annotated with metadata about the compliance, such as which regulation, standard, rules, compliance requirements, and so on, have led to the design and implementation of the control. The main goal of the Compliance Metadata model is hence to support the automatic documentation of all compliance-related concerns. In other words, the Compliance Metadata model allows us to not only model “how” compliance is achieved, but also “why”.

File: D1.2_Core-meta-models-transformation-templates-and-languages.doc Page 63 of 69
5.2. Compliance Metadata model

In this section we propose a compliance metadata model which serves as a bridge between the compliance concerns – represented in terms of aforementioned compliance DSLs and organisational functionality specified by VbMF process view models as well as compliance sources. The Compliance Metadata model correlates process view models with compliance metadata such as compliance documents, requirements, risks, and so on. On the one hand, such annotations can be used for facilitating automatic compliance controls [DoW]. That is, services or processes are distinguished by annotation to implement and realise automatic controls. On the other hand, the metadata serves as an information source for automatically generating reports and documentations of compliance requirements and implementations.

A compliance requirement may directly relate to an organisational unit such as a process, a service, or a business object. Nonetheless compliance requirements not only introduce new but also depict orthogonal concerns to these; although usually related to process-driven SOA elements, they are often pervasive throughout the SOA and express independent concerns. In particular, compliance requirements can be formulated independently until applied to a SOA. As a consequence, compliance requirements can be reused, e.g., for different processes or process elements.

Figure 42 shows a proposed conceptual model for expressing compliance metadata. The conceptual model shown in Figure 42 has been developed following an integration effort of the whole consortium. As a consequence, this model now serves as a basis for bridging underlying data models of COMPAS prototypes [D1.3, D2.6, D4.2, D4.4, D5.5]. By using the view extension mechanism presented in Section 3.5.1 various concepts provided by other WPs can be integrated into the MDSD Software Framework via the NamedElement class.
In Figure 42, the core concepts are Control, Compliance Rule, and Technical Control. A Control describes the restraining or directing influence to check, verify, or enforce rules to satisfy one or more compliance requirement -- at the business level. Thus, the notion of Control reflects the understanding and interpretation of laws, regulations, policies, etc. of domain and compliance experts. A Technical Control stands for a logical part of the business process that exercises the restraining or directing influence to check, verify, or enforce rules to implement a control. These concepts are bridged together via the notion of Compliance Rule. The data models of various WP prototypes, by extending and deriving from the aforementioned concepts, are smoothly integrated to shape a unified Compliance Metadata model.
5.3. Integration of related prototypes

Figure 43 presents an excerpt of the overall COMPAS architecture that depicts the integration of prototypes that are closely relevant to the MDSD Software Framework. We highlight the main integration points as shown in Figure 43:

1. Both VbMF and DSL tooling are smoothly integrated with the COMPAS repository in order to store process view models and view instances, DSLs representing business compliance, and compliance metadata describing the relationships between process models (i.e., process views) and business compliance (i.e., compliance DSLs) [D4.4]. As a result, the concepts of the compliance metadata model play important roles in supporting these integrations as presented in Section 5.2.

2. MDSD Software Framework, by the means of template-based code generations, is also integrated with the runtime infrastructure including application servers and process engines [D4.4]. MDSD Software Framework can generate not only the process implementations in terms of BPEL/WSDL descriptions but also the configurations and directives that support deployment, enactment, and monitoring of processes and/or compliance at runtime.

6. Conclusions

In this deliverable, we presented the MDSD Software Framework that provides models for the core concepts of COMPAS and domain-specific languages (DSL) for modelling various types
of models. At the heart of the MDSD Software Framework is a view-based, model-driven approach (VbMF) that offers different view models as well as clear extension points for the compliance concerns defined in other WPs such as regulations, security, licensing, QoS, etc. DSLs are means to describe knowledge via a graphical or textual syntax, which is tied to domain-specific modelling elements through a language meta-model. We define the foundations for DSL language design and illustrate how to formulate, extend and integrate DSLs and VbMF. In addition, we proposed a model transformation infrastructure based on template-based techniques to generate business process implementations and deployment configurations as well as supporting runtime directives for monitoring compliance. Last but not least, we presented the integration of the MDSD Software Framework and related prototypes developed by other WPs.

7. Reference documents

7.1. Internal documents

[DoW] “Description of Work” for COMPAS, final version of 2008-02-01.


7.2. External documents


