Master thesis

Multi-level views in a SUM-based environment

Peter Lanting

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Department of Computer Science - Software Engineering specialisation

EXAMINATION COMMITTEE

dr. L. Ferreira Pires
dr. ir. M.J. van Sinderen

January 21, 2015
Contents

List of Figures VI
List of Listings VII
List of Abbreviations IX
Acknowledgements XI
Abstract XIII

1 Introduction 1
1.1 Motivation ................................................................. 1
1.2 Objectives .................................................................. 2
1.3 Research questions ................................................... 3
1.4 Approach .................................................................. 3
1.5 Structure of the report .................................................. 4

2 Multi-Level Modelling 7
2.1 MDA limitations ......................................................... 7
2.1.1 Dual Classification .................................................. 8
2.1.2 Class/object Duality .................................................. 8
2.1.3 Replication of Concepts .......................................... 9
2.2 OCA principles .......................................................... 9
2.2.1 Clabject .................................................................. 10
2.2.2 Potency-based deep instantiation ......................... 11
2.3 OCA requirements ....................................................... 12
2.4 Visualization of PLM using LML ............................... 13
2.5 Conclusion ................................................................. 15

3 View- and SUM-based Modelling 17
3.1 View-based modelling ............................................... 17
3.1.1 Standard terminology .............................................. 18
3.1.2 Consistency ........................................................... 19
3.2 SUM-based modelling ................................................ 20
3.3 Multi-level SUM-based modelling ............................... 23
3.3.1 ATL Adapter .......................................................... 24
3.3.2 Multi-level transformations .................................... 25
3.4 Conclusion ................................................................. 26

4 Available tool support .................................................. 27
4.1 nAOMi ................................................................. 27
   4.1.1 Structure and functionality .................................... 27
   4.1.2 Current state of development ................................. 30
4.2 Melanee ............................................................. 30
   4.2.1 Structure and functionality .................................... 30
   4.2.2 Current state of development ................................. 32
4.3 Conclusion ............................................................. 32

5 Context view design .................................................... 33
5.1 View selection ......................................................... 33
   5.1.1 Sequence diagram ............................................... 34
   5.1.2 Context diagram .................................................. 36
   5.1.3 Selection .......................................................... 37
5.2 Context view implementation ....................................... 38
   5.2.1 Metamodel design ................................................ 38
   5.2.2 DSL implementation ............................................. 39
   5.2.3 SUM metamodel extension ..................................... 41
   5.2.4 Alternative representations .................................... 42
5.3 Structural view extension ........................................... 43
5.4 Conclusion ............................................................. 44

6 Model transformations .................................................. 45
6.1 ATL ................................................................. 45
6.2 Transformation chain ............................................... 46
6.3 Generate view transformations ..................................... 47
   6.3.1 Generate context view .......................................... 48
   6.3.2 Generate structural view ....................................... 50
   6.3.3 Generate helper view .......................................... 51
   6.3.4 Helpers .......................................................... 51
6.4 Merge transformations ............................................... 52
   6.4.1 Merge context view ............................................. 53
   6.4.2 Merge structural view .......................................... 54
   6.4.3 Merge helper view ............................................. 54
6.5 Merge SUM transformation .......................................... 55
6.6 Conclusion ............................................................. 56

7 Implementation .......................................................... 57
7.1 Eclipse plug-ins ....................................................... 57
   7.1.1 Transformation plug-in ......................................... 58
   7.1.2 SUM plug-in ...................................................... 59
7.2 Dimension configuration .............................................. 59
7.3 Conclusion ............................................................. 61

8 Case study ............................................................... 63
8.1 Overview .............................................................. 63
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1.1</td>
<td>Requirements</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Description</td>
</tr>
<tr>
<td>8.1.3</td>
<td>Actors</td>
</tr>
<tr>
<td>8.2</td>
<td>Constructing the context model</td>
</tr>
<tr>
<td>8.3</td>
<td>Constructing the structural model</td>
</tr>
<tr>
<td>8.4</td>
<td>Linking the views</td>
</tr>
<tr>
<td>8.5</td>
<td>Conclusion</td>
</tr>
<tr>
<td>8.5.1</td>
<td>Current limitations</td>
</tr>
<tr>
<td>8.5.2</td>
<td>Other cases</td>
</tr>
<tr>
<td>9</td>
<td>Conclusion</td>
</tr>
<tr>
<td>9.1</td>
<td>Limitations</td>
</tr>
<tr>
<td>9.2</td>
<td>Further research</td>
</tr>
<tr>
<td>References</td>
<td>79</td>
</tr>
<tr>
<td>Appendices</td>
<td>81</td>
</tr>
<tr>
<td>A</td>
<td>Struct_SUM2View</td>
</tr>
<tr>
<td>B</td>
<td>Context_SUM2View</td>
</tr>
<tr>
<td>C</td>
<td>Struct_View2SUM</td>
</tr>
<tr>
<td>D</td>
<td>Context_View2SUM</td>
</tr>
<tr>
<td>E</td>
<td>Merge_SUM</td>
</tr>
<tr>
<td>F</td>
<td>Helper_SUM2View</td>
</tr>
<tr>
<td>G</td>
<td>Helper_View2SUM</td>
</tr>
<tr>
<td>H</td>
<td>Dimension Configuration file</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 Report outline and chapter dependencies. .................................................. 5
2.1 The four-layer architecture of MDA. .......................................................... 8
2.2 The Orthogonal Classification Architecture. .............................................. 10
2.3 A clabject with a type and an instance facet [14]. .................................... 10
2.4 The PLM rendered using LML syntax [28]. ................................................. 14
3.1 Conceptual model of the Architecture Description [32]. ................................ 18
3.2 Worst case scenario: view-based approach vs. SUM-based approach [19]. ...... 20
3.3 Dimension-based navigation principle [19]. ............................................. 22
3.4 Dimension-based navigation in nAOMi. .................................................... 23
3.5 Illustration of a Single-Underlying Model and its views [18]. ....................... 23
3.6 A multi-level behavioural view, instantiated from a part of the KobrA metamodel [10]. . 24
3.7 Architecture of the multi-level ATL adapter. ............................................. 25
3.8 Transformation without explicitly stating * potency defined on component [11]. .... 26
4.1 Kobra-oriented SUM metamodel [10]. ....................................................... 28
5.1 SUM metamodel extension concept for a sequence diagram ....................... 35
5.3 DSL representation for a sequence diagram ............................................. 35
5.2 Example of a sequence diagram .............................................................. 36
5.4 Example of context diagram ................................................................. 36
5.5 Context metamodel concept ................................................................. 37
5.6 DSL representation for a context diagram ............................................. 37
5.7 The context metamodel ............................................................... 39
5.8 Context model based on the metamodel ............................................. 40
5.9 DSL visualization editor for system entity ........................................... 41
5.10 DSL representation of the context model. ............................................. 41
5.11 The currently implemented SUM metamodel ........................................ 42
5.12 The extended SUM metamodel ............................................................ 42
5.13 Helper view, used to connect subSystems with components ................... 43
5.14 Context model in specifiction(black box) mode ...................................... 43
5.15 The current structural view metamodel .................................................. 44
6.1 Visualization of the transformation chain ............................................. 46
7.1 Extension point definitions (left) and extensions (right) ......................... 58
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>New SUM project with one <em>system</em> entity.</td>
<td>66</td>
</tr>
<tr>
<td>8.2</td>
<td>Context view of the holiday booking system.</td>
<td>67</td>
</tr>
<tr>
<td>8.3</td>
<td>Structural view with one main component.</td>
<td>68</td>
</tr>
<tr>
<td>8.4</td>
<td>Structural view of the holiday booking system.</td>
<td>69</td>
</tr>
<tr>
<td>8.5</td>
<td>Linking sub-systems with components in the helper view</td>
<td>70</td>
</tr>
</tbody>
</table>
List of Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Multi-level ATL code</td>
<td>26</td>
</tr>
<tr>
<td>6.1</td>
<td>Context_SUM2View: Actor and RelationshipFrom rule</td>
<td>48</td>
</tr>
<tr>
<td>6.2</td>
<td>Struct_SUM2View: Component and Creates rule</td>
<td>50</td>
</tr>
<tr>
<td>6.3</td>
<td>Context_SUM2View: Helpers</td>
<td>51</td>
</tr>
<tr>
<td>6.4</td>
<td>Context_SUM2View: Helper createMethods</td>
<td>52</td>
</tr>
<tr>
<td>6.5</td>
<td>Context_View2SUM: Actor and RelationshipFrom rule</td>
<td>53</td>
</tr>
<tr>
<td>6.6</td>
<td>Struct_View2SUM: Component and Creates rule</td>
<td>54</td>
</tr>
<tr>
<td>6.7</td>
<td>Merge_SUM: toTransform helper</td>
<td>55</td>
</tr>
<tr>
<td>7.1</td>
<td>Part of the dimensionContainer in the configuration file</td>
<td>59</td>
</tr>
<tr>
<td>7.2</td>
<td>Example of a mapping in the configuration file</td>
<td>60</td>
</tr>
<tr>
<td>A.1</td>
<td>Struct_SUM2View transformation</td>
<td>83</td>
</tr>
<tr>
<td>B.1</td>
<td>Context_SUM2View transformation</td>
<td>87</td>
</tr>
<tr>
<td>C.1</td>
<td>Struct_View2SUM transformation</td>
<td>93</td>
</tr>
<tr>
<td>D.1</td>
<td>Context_View2SUM transformation</td>
<td>97</td>
</tr>
<tr>
<td>E.1</td>
<td>Merge_SUM transformation</td>
<td>101</td>
</tr>
<tr>
<td>F.1</td>
<td>Helper_SUM2View transformation</td>
<td>105</td>
</tr>
<tr>
<td>G.1</td>
<td>Helper_View2SUM transformation</td>
<td>107</td>
</tr>
<tr>
<td>H.1</td>
<td>Dimension configuration file</td>
<td>109</td>
</tr>
</tbody>
</table>
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATL</td>
<td>ATLAS Transformation Language</td>
</tr>
<tr>
<td>CAD</td>
<td>Computer-Aided Design</td>
</tr>
<tr>
<td>CBD</td>
<td>Component-Based Development</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain Specific Language</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modelling Framework</td>
</tr>
<tr>
<td>ER</td>
<td>Entity Relationship</td>
</tr>
<tr>
<td>GMF</td>
<td>Graphical Modelling Framework</td>
</tr>
<tr>
<td>GPL</td>
<td>Graph Product Line</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>IEC</td>
<td>International Electrotechnical Commission</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organisation for Standardization</td>
</tr>
<tr>
<td>LML</td>
<td>Level-agnostic Modelling Language</td>
</tr>
<tr>
<td>MDA</td>
<td>Model-Driven Architecture</td>
</tr>
<tr>
<td>MDE</td>
<td>Model-Driven Engineering</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta Object Facility</td>
</tr>
<tr>
<td>OCA</td>
<td>Orthogonal Classification Architecture</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>OSM</td>
<td>Orthographic Software Modelling</td>
</tr>
<tr>
<td>OWL</td>
<td>Web Ontology Language</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform-Independent Model</td>
</tr>
<tr>
<td>PLE</td>
<td>Product-Line Engineering</td>
</tr>
<tr>
<td>PLM</td>
<td>Pan-Level Model</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform-Specific Model</td>
</tr>
<tr>
<td>PVS</td>
<td>Prototype Verification System</td>
</tr>
<tr>
<td>SUM</td>
<td>Single-Underlying Model</td>
</tr>
<tr>
<td>SW</td>
<td>Semantic Web</td>
</tr>
<tr>
<td>SWT</td>
<td>Standard Widget Toolkit</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>VOSE</td>
<td>Viewpoint-Oriented Software Engineering</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>XSD</td>
<td>XML Schema Definition</td>
</tr>
</tbody>
</table>
Acknowledgements

During the time span of my master thesis I have received a lot of support from many people that I would like to thank here. First of all I would like to thank my supervisor Luís Ferreira Pires, who inspired me to conduct my research in the field of multi-level and SUM-based modelling, and always supported me during my research and writing. I would like to extend my gratitude to my second supervisor Marten van Sinderen, for the support and comments on my work and thesis. Furthermore, I would like to show my gratitude to Christian Tunjic and Ralph Gerbig from the University of Mannheim, for all the support regarding the tools Melanee and nAOMi, which I used during my thesis. Whenever I had problems or questions they found some time to help me, which I greatly appreciated.

Besides, I would like to thank my family, for all the emotional and financial support throughout my study and my life, without them I would be nowhere. Special thanks go to my girlfriend Lily, for all the love and support she gave me throughout the process. Last but not least I would like to thank my friends for all the good times we have had together.
Abstract

In this thesis we investigate the applicability of combining multi-level modelling and Single-Underlying Model (SUM)-based modelling. Multi-level modelling is a modelling methodology that solves the limitations of the traditional Model-Driven Architecture (MDA). Multi-level modelling acknowledges the difference between ontological and linguistic entities, and supports an unlimited amount of ontological levels, whereas MDA does not make this distinction and only supports a four-level architecture. SUM-based modelling is a branch of view-based modelling that uses one main model to store all the information about a system. From this SUM, different views can be generated on demand. Each view shows a specific part of the system, which can be edited and merged with the SUM, to save the changes.

During this process we also looked into the available tools and found Melanee, a tool that supports the design and transformation of multi-level models, and nAOMi, a tool that supports SUM-based modelling and uses Melanee for the view visualization. nAOMi is still in an early stage of development, and only supported a partial structural view. This inspired us to design an extension for nAOMi that implements an additional view, using both multi-level modelling and SUM-based modelling methodologies. This process consists of several steps such as the selection and design of the view (meta)model, the transformations to generate and merge the view, and the implementation of a modular extension to nAOMi. We chose to implement a context view, because it adds new possibilities to SUM-based modelling in nAOMi, has a clear connection with the currently partly supported structural view, and can be rendered in a custom DSL visualization using the DSL editor of Melanee. Furthermore, we extended the partially supported structural view to a completely supported structural view. For both these views we developed ATL transformations that allow the generation from the SUM, as well as merging the edited views with the SUM. nAOMi is built as an Eclipse plug-in and provides several extension points, which allows us to extend the tool with new views and SUM models. This guarantees that the original code is not changed and thus meets our requirement to provide a modular extension.

In order to test our extension we designed and conducted a case study that resembles the design of a real software system. The results show that there are still some limitations, in both nAOMi and our extension, that mainly concern the visual aspects and rendering of the views. However, all the implemented functionality works, and our extension provides a useful addition to nAOMi.
Chapter 1

Introduction

1.1 Motivation

State-of-the-art methodologies for software design require the use of modelling techniques. The most popular modelling techniques are based on the Model-Driven Architecture (MDA), introduced by the Object Management Group (OMG) in 2001 [35]. Modelling techniques include two different techniques, namely the actual (meta)modelling design of software systems, and the support for model transformations. These transformations enable platform-independent models to be transformed to platform-specific models, or vice versa. Although MDA is the standard for modelling, it still has some limitations and inconsistencies, especially with the metamodelling layering on which it is based. For example, it lacks the ability to carry data across multiple levels. In order to solve these limitations new (meta)modelling techniques were proposed [15]. These modelling techniques, now known as multi-level modelling, led to the development of a new architecture, the Orthogonal Classification Architecture (OCA) [12]. In order to enable the utilization of multi-level modelling, the tool Melanee [33] was developed. This tool supports the design of multi-level models, as well as the transformation of these models to other multi-level, or MDA-based, models [9].

Another modelling approach presented in the literature is Single Underlying Model (SUM)-based modelling [18], which is a branch of view-based modelling. View-based modelling enables the design of rather complex systems by representing the system in several sub-models, called views. Each view describes a specific aspect of the system, such that the overall complexity can be controlled [20]. SUM-based modelling is based on this approach, but instead of using multiple separate views to describe the system, one main model, the SUM, is used to generate these views. By using this technique all the information about the system is stored in the SUM, and the different views can be generated on-demand by the users. These views can then be modified, after which they need to be synchronized with the SUM in order to keep all views consistent. A tool that supports the development of SUM-based modelling is nAOMi [34].

Atkinson et al. proposed the idea of combining multi-level modelling with SUM-based modelling [10]. By developing a new version of nAOMi that supports multi-level modelling, a foundation for developing multi-level SUM-based models was established. In order to represent the multi-level views, nAOMi uses the tool Melanee. Melanee supports the visualization of multi-level models, and also enables the utilization of custom representations. However, the new version of nAOMi is still in an early development stage, and
only the structural view of systems is currently supported. In order to contribute to the development of this new field of modelling the tooling needs to be extended with more functionality and views. This project is inspired by this new modelling approach, and the rudimentary tooling that is available. Therefore, during this thesis project, we implement a view as an addition to nAOmi.

1.2 Objectives

The goal of this thesis project is to demonstrate the feasibility of multi-level SUM-based modelling, by extending the existing SUM-based modelling tool nAOmi. Hereby we have also identified the practical limitations that this modelling approach bears in its current state. nAOmi is a SUM-based tool that initially only worked with UML models, but now has a version that also supports multi-level modelling. The tool currently only supports a part of a structural view, and is under development for the complete structural view and a behavioural view. In this project we identified which view would bring a useful addition to the tool, and added this view accordingly. The view is represented in a multi-level environment which is supported by the multi-level model development tool Melanee. This tool has been used to create and represent multi-level views. Using a graphical DSL editor, the view representation can be adjusted such that the view looks like its original design. Since nAOmi is a SUM-based modelling tool, the view should be able to be generated from the SUM, and should be able to merge with the SUM. This required the development of multi-level transformations between the SUM and each view.

Design requirements

The design requirements are the specifications to which the extension should conform. These requirements below show the boundaries of the additional functionality that we have added to nAOmi:

- Design a new view that:
  - has unique capabilities that are not covered by the currently implemented views.
  - has a logical connection to the current SUM metamodel.
  - has a DSL representation that closely resembles the standard visualization of this view.
- Extend nAOmi without affecting the currently available code.
  - Add functionality to show a complete overview of the system in the selected view.
- (Optionally) extend the partly implemented structural view to a fully functioning view.
1.3 Research questions

From the objectives, the following research questions can be defined:

- **RQ1:** What is the applicability of multi-level SUM-based modelling?
  - **SQ1.1:** What is the coverage of multi-level modelling w.r.t. our design requirements?
  - **SQ1.2:** What is the coverage of SUM-based modelling w.r.t. our design requirements?
  - **SQ1.3:** How stable and complete are the available tools?

- **RQ2:** How can we extend nAOMi with a view in a modular way?
  - **SQ2.1:** How do we extend the SUM metamodel without effecting the original metamodel?
  - **SQ2.2:** How do we represent the view in a DSL visualization?

Two main research questions have been defined, the first being a knowledge question and the second being a design question. The first research question (RQ1) concerns the feasibility of multi-level SUM-based modelling in its current state and support. Hereby, we look into the maturity and current state of the multi-level modelling methodology (SQ1.1), the maturity of SUM-based modelling methodology (SQ1.2) and the maturity of the tools that are currently supporting multi-level modelling as well as SUM-based modelling (SQ1.3). The second research question (RQ2) asks for an implementation of a modular extension to the tool nAOMi. In order to create a modular extension we need to find a way to extend the current SUM metamodel without overwriting the original one (SQ2.1). Furthermore, we should find a manner to represent the extended view in its standard DSL visualization (SQ2.2).

1.4 Approach

The approach we have taken in this project consists of the following steps:

1. Study existing views and choose one to extend the tool with.

2. Explore the tools nAOMi and Melanee.
   - Study the source code of both tools.
   - Study the core structure of the KobrA-based SUM in nAOMi.
   - Identify the interoperability between nAOMi and Melanee.

3. Design and implement a simple core addition to the KobrA metamodel in order to support the view.
4. Create the graphical representation for the view using the DSL editor in Melanee, and identify how to implement this representation using code.

5. Create the multi-level transformations between the Kobra-based SUM and the view.

6. Implement the view as an extension to nAOMi.
   - Figure out how to generate the view from the SUM.
   - Figure out how to merge the view with the SUM.

7. Design a case study that shows the applicability of the extended view.
   - Test the implementation with this case study.

1.5 Structure of the report

The remainder of the report is structured as follows. Chapter 2 elaborates on multi-level modelling. Chapter 3 discusses the concepts of view-based modelling and SUM-based modelling, including the proposed idea of combining SUM-based modelling with multi-level modelling. Chapter 4 focusses on the tools nAOMi and Melanee, their core structure and their interoperability. Chapter 5 introduces the extended view and its graphical representation. Furthermore, the changes to the KobrA metamodel are discussed in this chapter. Chapter 6 discusses the multi-level transformations necessary to generate the views from, and merge the views with the SUM. Chapter 7 examines the implementation of the extension to nAOMi. Chapter 8 discusses a case study to demonstrate the applicability of the approach and the suitability of our tool extension. Chapter 9 concludes this thesis project. Figure 1.1 shows the main dependencies between these chapters.
Figure 1.1: Report outline and chapter dependencies.
Chapter 2

Multi-Level Modelling

Multi-level modelling is an approach to modelling whereby multiple ontological classification levels are covered within one model. This approach is used to solve some limitations of the classical Model-Driven Architecture (MDA) [35]. In this chapter these limitations and their solutions regarding multi-level modelling are discussed. Furthermore this chapter elaborates on the Pan Level Model and the Level-agnostic Modelling Language, which implement the Orthogonal Classification Architecture.

2.1 MDA limitations

The Object Management Group (OMG) fostered the development of the Model-Driven Architecture (MDA), in order to create an architecture that achieves interoperability, portability and reusability of platform-independent models. MDA is based on two main techniques, namely, metamodelling and model transformation. The former can be used to design models of software systems, and the latter can be used to transform these models to, e.g., platform-specific models, or vice versa. MDA is built on the principles of object-oriented modelling, and uses a four-layer metamodelling architecture. The top level M3 is the Meta-Object Facility (MOF) level, which is the metamodelling language to define metamodels at the M2 level. A widely used metamodel language is the Unified Modelling Language (UML) [26], which on its turn can be used to build user models on M1. M0 represents the real world instances of the model, as it can be seen in Figure 2.1.

An important principle that is used within the MDA is strict modelling. Strict modelling implies that every element of a model on level N must be an instance of exactly one element on level N+1. All relationships other than instance-of relationships between two elements must be defined on the same level. Using strict modelling results in many benefits, since it makes it much easier to type-check and therefore debug programs or models. This contributes to a rather simple design of models. Loose modelling is the opposite of strict modelling, and allows all relationships to cross multiple levels and to connect to multiple elements. This usually results in rather complex models that are hard to understand. Besides this, loose modelling increases the chance of defining logical inconsistencies. So although strict modelling gives us some limitations towards the relationships between elements, it also gives great benefits for the consistency and the relative simplicity of the models.
Strict modelling is one of the underlying reasons for the current limitations in MDA. These limitations of MDA have been identified, and solutions have been proposed in [8] [15] [5]. Almost all of these limitations can be found on the M3 level, where the MOF is defined. The most important limitations are discussed below.

![Figure 2.1: The four-layer architecture of MDA.](image)

### 2.1.1 Dual Classification

One of the main problems with MDA is dual classification. On level M1 in Figure 2.1, there is an element named `aVideo` with an attribute `title` that holds the value *Casablanca*. Next to this element there is another element named `Video` with an attribute `title` of type String. Normally speaking, `aVideo` would be an instance of `Video`. However, Figure 2.1 only shows that `aVideo` is an instance of the linguistic `Instance` on the M2 level. This is because MDA only uses linguistic instantiations. So even though `aVideo` is actually an instance of two elements, namely the ontological `Video` and the linguistic `Instance`, only the linguistic instance relation is supported. In order to model the ontological instantiation in MDA, workarounds have to be used. In Figure 2.1 a stereotyped dependency named `snapshot` is used to represent the instance relation between `Video` and `aVideo`.

### 2.1.2 Class/object Duality

Another problem with MDA is the Class/Object duality problem. In the MOF there are two kinds of elements namely, types and instances. Looking at Figure 2.1, `Video` instantiates from `Class` and `aVideo` from `Instance`. This means that `Video` is the type and `aVideo` is the instance. This makes perfect sense, however, when we introduce an element that represents the type of a type, this could cause some problems. Imagine that in Figure 2.1 we introduce an element named `Media` on the M1 level, from which `Video` is ontologically
2.2. OCA PRINCIPLES

instantiated (like aVideo is ontologically instantiated from Video). Looking from the point of view of Media, Video should now be an instance rather than a type. This introduces a problem, since Video should have the properties of a type and an instance at the same time, which is not possible according to the MOF. Similar to the previous problem, workarounds have been provided in order to cope with this problem, such as stereotypes and powertypes [25].

2.1.3 Replication of Concepts

Replication of concepts arises when information should be transferred across more than one instantiation level. In UML it is only possible to influence elements on the next instantiation level, e.g., from M3 to M2. So in order to influence over more than one level it is necessary to duplicate this information over multiple levels. This leads to unnecessarily complex models, with the same data replicated by different elements. Imagine, for example, that there are some properties of the Class type on the M3 level of Figure 2.1 that should influence the properties of Video at the M1 level. Since information can only influence the direct adjacent level, this is not possible without specifying the same information again on level M2.

2.2 OCA principles

For most of the MDA limitations, workarounds have already been provided, but even though they do their job, they bring unnecessary complexity to the models. This leads to more time being used to develop the models, and to additional techniques that have to be learned by the developers in order to create models. Therefore, an architecture that solves these limitations has been developed, namely the Orthogonal Classification Architecture (OCA), which supports multi-level modelling. The theoretical foundations of OCA have been proposed by Atkinson and Kühne [12]. Several other researchers contributed to these proposals by creating NIVEL, which is a formal foundation introduced by Asikainen and Männistö [4], or by using real cases in practice, like Aschauer et al. [3]. Like MDA, OCA uses the principles of strict modelling as a foundation for creating models. As we discussed above, strict modelling gives great benefits regarding consistency and the ease of understanding and developing models. Figure 2.2 shows the same model as Figure 2.1, but now according to OCA.

The main difference that immediately stands out in Figure 2.2 is that this architecture does not use one dimension with four M levels like MDA, but it uses two dimensions each according to the principles of strict modelling. These dimensions are represented by L, for the linguistic dimension, and O, for the ontological dimension. The linguistic dimension consists of three different levels: the top level L0 contains the language definition, which is the Pan-Level Model (PLM); level L1 corresponds to the M1 level of the MDA and contains the models created by the user using instantiations from L0; and the final level L2 represents the real world instances of the elements created at L1. The number of ontological levels is not fixed, and theoretically it can range from one to an unlimited amount of levels. All the ontological levels are present at the L1 level of the linguistic dimension. Within the ontological dimension, the O0 level represents the main elements from which other elements at the O1 level can be instantiated. O1 contains types that can be instantiated at O2, and so forth, until the lowest ontological level is reached.

The advantage of having a two-dimensional classification hierarchy is that all elements can now be instan-
CHAPTER 2. MULTI-LEVEL MODELLING

10

tiated from both an ontological type and from a linguistic type. This structure immediately solves the dual classification problem described in Section 2.1.1. Without any workarounds, and using a simple and clear relationship, a Video can now be instantiated from both Video as well as Instance. Unlike UML, OCA combines types and instances into one element called Clabject, which is discussed in the next section. Therefore, in Figure 2.2 Video is instantiated from Clabject, which is both a class and an instance.

![Figure 2.2: The Orthogonal Classification Architecture.](image)

2.2.1 Clabject

A key element to enable the utilization of multiple ontological levels is the clabject. The term clabject is originated from the combination of class and object, due to the roles that it combines. In MDA there was a distinction between a type and an instance element, where in OCA the clabject combines both of these elements. In order words, clabjects have both a type facet as well as an instance facet. A visualization of a clabject is shown in Figure 2.3, with on the right hand side the instance facet, and on the left hand side the type facet. Clabjects can also contain attributes and method types, and attribute values and method instantiations, depending on the facet(s) it offers.

![Figure 2.3: A clabject with a type and an instance facet.](image)

As we showed in the example of the class/object duality problem in Section 2.1.2, in MDA it was impossible for Video to be both a type as well as an instance. With the use of clabjects in OCA, Video can now have a
2.2. OCA PRINCIPLES

2.2.1 Type facet for aVideo on the right, and an instance facet for Media on the left. This solves the class/object duality problem and enables the utilization of multiple ontological levels.

2.2.2 Potency-based deep instantiation

In order to solve the replication of concepts problem, it was necessary to find a way to pass on information across multiple levels. Therefore Atkinson and Kühne introduced the concept of potency-based deep instantiation [8]. Deep instantiation means that elements can be instantiated over more than one level, and the potency helps keep track of the number of instantiations. Every clabject can have a potency, which is a positive integer value representing the number of instantiations it allows. The value of the potency can be classified in three categories:

- **Potency 0.** With a potency of zero the clabject or attribute cannot be instantiated. This also implies that if a clabject has a potency zero, it corresponds to an abstract class, since it cannot be instantiated.

- **Potency 1 or more.** Potency with a value of 1 or higher indicates that the clabject or attribute can be instantiated. Depending on the value n of the potency it can be instantiated n times. If, for example, clabject x instantiates clabject y, y must have a potency of exactly one lower than the potency of x. For example, if the potency of x is one, the potency of y is zero.

- **Potency *.

Besides clabjects, the attributes and values of the attributes also have potencies. To prevent any confusion we use the word **potency** for clabjects, **durability** for attributes and **mutability** for the value of attributes. All these potency values obey the standard rules mentioned above, however, there are a few differences between them. Durability indicates the number of instances of the clabject it belongs to that also have this attribute. For example, clabject A has a potency of two and an attribute x with durability one. Clabject B is an instance of clabject A and clabject C is an instance of clabject B. Since x has a durability of one, it means that clabject B has an attribute x with durability zero, which then means that clabject C does not have attribute x. Furthermore the durability of zero indicates that the attribute should have a value at this level. The durability is not allowed to be higher than the potency of its corresponding clabject. This is a logical conclusion from the boundedness of a clabject, and thus cannot exist without one.

**Mutability**

Mutability is a subcategory of the potency concept. The mutability indicates whether the value of an attribute can be changed or not. A mutability of zero indicates that the value of an attribute cannot be changed and must retain the same value as the corresponding attribute of the clabject from which the clabject is instantiated. So when an attribute with durability two and mutability zero is created, it means that the value of this attribute is fixed for all the instantiations of the clabject corresponding to the attribute. When the
mutability is one or higher, the value can be altered if desired. Every time the clabject with this attribute is
instantiated the mutability decreases with exactly one, as with the durability. The mutability is not allowed
to have a higher value than the attribute durability, and it can only have a * mutability when the durability is
also *.

2.3 OCA requirements

The language used to describe OCA comprises two syntaxes, namely a concrete syntax named the Level-
agnostic Modelling Language (LML) and an abstract syntax named the Pan-Level Model (PLM). Together
they form the foundation for the implementation of OCA, which supports the multi-level modelling prin-
ciples. LML was developed by Atkinson et al. [13] and PLM by Kennel [28]. PLM provides the fundamental
metamodel from which models on the L1 level can be created. In order to develop the language that imple-
ments OCA, several requirements have been set. The most important requirements are discussed below.

- **UML-like**

  Although UML has some limitations, it is has become the standard modelling language for creat-
ing graphical software models. Software engineers are familiar with the UML representation and
therefore LML was designed to use a similar visualization to UML, such that it would be easy for
developers to switch to LML without the need to learn a complete new modelling language. Besides
the visual aspects of UML, the abstract concepts used in PLM are also designed to work similarly to
UML, like the use of classes and objects.

- **Level-agnostic**

  LML and PLM are based on the concepts of the OCA, and therefore should include all the multi-
level modelling features. In UML different elements and their connections can be represented in
different ways, e.g., classes and objects. LML and PLM represent these elements and connections in
a uniform way. Being level-agnostic basically means that the language should support the uniform
representation of all elements across the different ontological levels.

- **Include different modelling paradigms**

  Besides UML, which is the standard language in software engineering, there are more modelling
paradigms from different communities. Examples include the Web Ontology Language (OWL) [27],
for the Semantic Web (SW) community, and the Entity Relationship (ER) modelling languages for
the database community. One of the goals of the OCA language design was to support these mod-
delling paradigms, but respecting the restrictions of being UML-like. Together, the SW and database
communities can be considered as the knowledge engineering community.

The main differences between the software engineering and knowledge engineering communities are
their goals. Knowledge engineering particularly aims at creating new correlations between already
existing elements, by studying these elements separately. Software engineering however, aims at
defining new elements and create instantiations from these elements. By supporting both software
engineering as well as knowledge engineering techniques, a unified set of techniques can be used
within a single language framework. This allows the utilization of techniques for different fields
of modelling. For example, using reasoning services from the knowledge engineering community,
during model development in the software engineering community.
• **Include reasoning services**
  Reasoning services come from the knowledge engineering community and are convenient for model development and model checking. OWL is a modelling language that supports reasoning services. Because reasoning services are convenient for model development and checking, PLM and LML implement a foundation to support these services. A reasoning service can be, for example, a service that allows the user to add all instantiation relationships to an element with a single action. Other possibilities include checking the consistency of the models during development.

• **Straightforward**
  The language should be straightforward to use. Supporting all requirements in this section is not enough if the resulting language is too complex to use and understand. Therefore the language aims to use as few elements as possible, such that it is easy to use and understand. At some points this goes at the expense of the efficient implementation of the PLM, but this was the trade-off that had to be made.

These key requirements resulted in a straightforward language that is used to represent both the concrete and the abstract syntax for a multi-level modelling foundation based on the OCA principles. This language has the look and feel of UML, but also combines both the software engineering and knowledge engineering communities, by e.g. supporting reasoning services. PLM models are represented using the LML concrete syntax, used in Figure 2.4, which is discussed in Section 2.4.

### 2.4 Visualization of PLM using LML

The PLM is the abstract syntax and the LML the concrete syntax of the OCA. Figure 2.4 shows the PLM metamodel with all its aspects in an LML representation. Compared to the concrete syntax of UML, it directly stands out that there is no clear hierarchical view, but using LML it is rather easy to spot the important elements of the system. Looking at Figure 2.4 it is directly visible that the clabject plays a central role in the PLM, but using UML this would not necessarily be visible. Besides this, the PLM uses less concepts to visualize the same amount of information, making it easier to use than UML. All the specific details on the different aspects of the PLM are discussed in [28]. This section elaborates on the concrete syntax defined in the LML, using Figure 2.4 as an example. Since research on this topic is still ongoing, the implementation shown in Figure 2.4 might slightly differ from the currently implemented version.

**DeepModel**
The deepModel is the main component in which everything is drawn. It is visualized as a rectangle with round corners, with the name of the ontology written in the right top corner. In Figure 2.4 this is the outer most rectangle named *PLM*.

**Level**
The level is a compartment in which the models are created, and represents an ontological level. The level is drawn within the deepModel and is represented as a rectangle. A deepModel can have multiple levels, which are separated by a horizontal line. Every level has its name written in the left top corner. The deepModel in Figure 2.4 has only one level, which is the *Linguistic Model*. 
Entity
An entity is the main component within the model, and is similar to the UML class element. The entities are visualized as rectangles with three compartments separated by horizontal lines. On the top there is the header, in the middle the attributes, and the methods are shown in the bottom. Examples of Entities in Figure 2.4 are Clabject, Connection and Feature.

Attribute
Attributes are defined in terms of a name and datatype, with optionally their value, durability and mutability. In Figure 2.4, examples of attributes can be found in the mid compartment of Clabject.

Method
Methods, like attributes, have a textual representation within the entities. Methods are shown in the bottom compartment and contain only the method signature, like "order() : int" in the entity Connection in Figure 2.4.

Connection
Connections can be represented in two different ways: (1) as a hexagon with the exact same three parts as the entities, or (2) as a small square dot. The latter is used in Figure 2.4, which gives a concise representation, but does not show the detailed information about the connections. Connections are used to describe links...
between entities.

**Role**
Roles connect entities to connections. A role is represented by lines with optionally a black arrowhead in the end, depending on whether the role is navigable. Furthermore, the multiplicity and the name can be represented close to the end of the line, as shown in Figure 2.4 e.g. between *Role* and *Connection*. The name of the role is also displayed above the line, whereas the multiplicity is shown below the line.

**Generalization**
A generalization can be represented as either an octagon or as a small square dot, similarly to connections. Generalizations indicate the subtype / supertype relationship between entities. In order to indicate which entity is the supertype, an empty triangle is shown at the end of the supertype. A generalization should have at least (but not limited to) one subtype and one supertype. An example of a generalization in Figure 2.4 is the relation between *Attribute*, *Method*, and *Feature*, where *Attribute* and *Method* are both subtypes of the supertype *Feature*.

**Classification**
A classification indicates an instance-type relation between entities, and can therefore cross the model boundaries. Classifications are represented by dashed lines with an empty triangle at the end of the type entity.

**SetRelationship**
A setRelationship describes the relation between two entities. They are represented by a rectangle with the top and bottom lines slightly curved. There are three different types of setRelationships, namely complement, equal and inversion. To indicate the type of relationship one of these three names is included in the rectangle.

**Potency**
Potency is expressed by giving the value of the potency as a superscript on the right side of the name. As we mentioned before, the potency should either be an integer or a *. The same visualization and rules apply for the durability and the mutability.

### 2.5 Conclusion

In this chapter multi-level modelling has been discussed. Traditionally MDA was used to model software systems, however over the years some MDA shortcomings have been identified and the community searched for workarounds and solutions. One of those solutions was multi-level modelling as introduced by Atkinson et al. [8]. We discussed the solutions that multi-level modelling brought, with respect to the main limitations of the traditional MDA. These solution comprised of multiple ontological levels, clabjects and potency-based deep instantiation. Based on these solutions, a new architecture (OCA) was developed. OCA comprises two syntaxes; the concrete syntax, LML, and the abstract syntax, PLM. Both LML and PLM have been discussed, including their goals and the different fields of model engineering that are supported. Finally the visual representation of the PLM is discussed in terms of the concrete syntax offered by LML.
Chapter 3

View- and SUM-based Modelling

In this chapter we discuss the concepts of view-based modelling and Single-Underlying Model (SUM)-based modelling. We look into the current state of these modelling techniques, and the difficulties that arise when using them. Furthermore we look into the differences between them, and investigate the idea of combining SUM-based modelling with the multi-level modelling techniques.

3.1 View-based modelling

Modelling software systems can be a complex activity. Especially with large systems, there are often multiple experts in different fields that have different responsibilities, concerns and roles with respect to the system. The concerns of these experts often overlap or intersect, and is often represented in different perspectives. Developing a system with all these experts can therefore be rather hard, if only a single representative model is constructed. It would be much more beneficial if each expert could work on his/her own part of the model, having only to deal with his own field of expertise. Therefore, in the early 90’s a new modelling approach was introduced, namely view-based modelling [23]. View-based modelling does not use one single model to develop a system, but uses several smaller models, the so called views. Each view represents a different aspect of the system, which enables experts to work only in their field of expertise without having to care about other aspects of the system. This enables faster and more efficient modelling of systems.

The first view-based modelling technique, developed in the early 90’s, is known as Viewpoint-Oriented Software Engineering (VOSE). VOSE is based on viewpoints that were introduced by Finkelstein et al. [22]. A viewpoint represents a specific aspect of the system, which is important to a specific developer. Each developer of the system has a different viewpoint based on his/her role and domain.

Based on this approach, several different view-based modelling frameworks have been introduced over the years. Among those are RM-ODP [21], ArchiMate [29] and Zachman [37], all using their own implementations of view-based software engineering. Since systems become increasingly larger and more complex, the need to separate the system in different views has greatly increased. Therefore, it also became increasingly important to define a standard terminology for view-based software development, such that the current and
future frameworks would be based on a common view-based technique. In 2000 the first standard definitions were documented in the IEEE 1471-2000 standard, which standardized the concepts of views and viewpoints, and how they should be implemented in a view-based system. This standard has been adopted by ISO/IEC as ISO/IEC 42010:2007 in 2007 and has been superseded by the currently used ISO/IEC/IEEE 42010:2011 [32].

3.1.1 Standard terminology

Figure 3.1 shows the conceptual model of the architecture description on which the requirements of ISO/IEC/IEEE 42010 are based. The most important aspects of this model and thus of the view-based modelling principles are the following:

- **Architecture Description** The architecture description always describes only one system and can be expressed in several ways, such as a document or a set of models. Furthermore it includes extra information that is determined by the organization, like authors, reviewers, change history, references, scope etc. Figure 3.1 shows that the architecture description plays the central role in this model and contains information about all the linked elements.

- **Stakeholder & Concerns** Every system has stakeholders that all have different interests and concerns with respect to the system. Examples of stakeholders and concerns can be developers interested in functionality or behaviour, or managers interested in the development or operational costs.

- **Architecture viewpoint** Architectural viewpoints are used to address the concerns of the stakeholders.
3.1. VIEW-BASED MODELLING

Each concern identified by any of the stakeholders must be addressed by at least one viewpoint. Viewpoints can also contain correspondence rules or methods for checking the consistency between models or their completeness.

- **Model kind** Model kinds define the conventions of one type of architectural model. It defines the languages, notations, modelling techniques, etc. used in this kind of model. Each viewpoint can have several model kinds to address its concerns in the most suitable way.

- **Architecture views** For each viewpoint there is exactly one architecture view, which expresses a concern of one or more of the stakeholders. This can be, for example, a behavioural view of the system-in-interest. All views have to adhere the conventions specified by its viewpoint, and they include one or more architecture models. Furthermore, they can contain information such as known issues with respect to its viewpoint, for example, known consistency conflicts.

- **Architecture model** An architectural model is created by adhering to the conventions described in its model kind. An architectural model must always comply to a model kind. With the use of architecture models, multiple notations within a view can be used, or information between different views can be shared. Models are not bound to one view, and can thus also be part of multiple views.

3.1.2 Consistency

One of the largest difficulties in view-based modelling is to keep the consistency between the different views. For each view that is changed and submitted, there is a need to check whether the other views in the same scope are still consistent with respect to the updated view. The rules and methods to check the consistency are defined in the viewpoints and determine what to do when inconsistency occurs. Especially with large systems, it is rather hard to maintain the consistency since it usually has more views, and thus it likely to have more overlapping views. For each overlapping view, a consistency check should be created and performed. Unlike the concepts described above, there is no common agreement about the best method for consistency checking between the different views. Therefore, multiple methods are being used by different tools. However, most of these methods translate views to logic to check the consistency. For example, Lopez-Herrejon and Egyed describe how they use propositional logic to check consistency between views, using a case study based on a Graph Product Line (GPL) program [30]. Besides checking consistency with logic, there are other methods, like metamodel-based consistency checking. In this case, a metamodel is defined to represent the features of the model. This metamodel is then checked against the views. This method is explored and tested with a case study by Paige et al. [31]. In this study, Prototype Verification System (PVS) and Eifel were used to define the metamodels to check which language is the most suitable for this purpose. The languages were compared regarding completeness and automation, and the authors concluded that Eifel is easier to use and has better automated consistency checking, while PVS is more expressive and offers better completeness.
3.2 SUM-based modelling

SUM-based modelling is closely related to view-based modelling. The main difference lies in the way the system is modelled. In view-based modelling, the system is defined in different views all linked together, whereas SUM-based modelling uses one single-underlying model to which all views are linked. One big advantage of using a single-underlying model is that there are fewer consistency checks to be performed. Each view has to conform to the main model only, and no longer to each view it overlaps with. Since consistency checks are one of the most difficult parts of view-based modelling, reducing consistency checks highly improves the development process.

![Figure 3.2: Worst case scenario: view-based approach vs. SUM-based approach [19].](image)

In order to give an example of how much clearer the system development looks like when using the SUM-based approach, Figure 3.2 shows a comparison of both methods. In both methods, the system is represented in five different views. On the left side we see the view-based approach, where each view has to be consistent with the views it overlaps with. Figure 3.2 shows a worst case scenario, where a lot of views overlap, leading to a lot of connections. On the right side we see the SUM-based approach, which only requires two connections for each view. This potentially facilitates the system development, since it reduces the amount of consistency checks that should be performed.

Orthographic Software Modelling

The Orthographic Software Modelling (OSM) is a SUM-based modelling approach introduced by Atkinson et al. in a series of papers [17] [18] [19]. OSM was developed to support a methodology that uses flexible view generation and management. OSM uses the KobrA 2 [6] method as a foundation for the SUM-based modelling framework.
Kobra 2

The original version of KobrA [16] was developed to combine three different techniques, namely Model-Driven Engineering (MDE), Component-Based Development (CBD) and Product-Line Engineering (PLE). Each technique represents one of the main dimensions that cover specific aspects of the system. MDE represents the abstraction dimension, CBD the composition dimension and PLE the projection dimension. Kobra 2 added two more dimensions, namely the encapsulation dimension and the variant dimensions. This leaves us with the following five dimensions:

**Abstraction** The abstraction dimension addresses the level of abstraction of a certain component, by selecting a view in this dimension. Although the views are not limited, there are three main views representing the Platform-Independent Model (PIM), the Platform Specific Model (PSM) and the implementation.

**Composition** The composition dimension addresses the specification of the selected elements. It includes the (de)composition of the selected elements as well as the sub-elements.

**Projection** The projection dimension addresses the kind of information displayed in the view. This can be, for example, structure or behaviour.

**Encapsulation** The encapsulation dimension addresses whether the selected component is viewed in a public (black box) or a private (white box) view. The public view contains all the information about external properties and can thus also be used as a requirements specification. The private view contains all the internal properties as well as the external properties of the public view. In OSM the public view is known as specification view, whereas the private view is known as realization view.

**Variant** The variant dimension addresses the different variants of a system. This can be, for example, a Windows-version or a Mac-version. Besides this, it also includes the decision model and all possible features of the selected variant.

**OSM principles**

Besides the dimensions that are present in KobrA, OSM added two dimensions to enable more specific views of the system: the granularity dimension and the operation dimension.

**Granularity** The granularity dimension addresses the distinction between type granularity and service granularity. Type granularity contains views that describe the types used by components, whereas service granularity focuses more on the required and provided interfaces.

**Operation** The operation dimension enables the selection of the different operations (methods) present in each component.

**Dimension-based View Navigation**

Together with the five dimensions of KobrA, a total of seven dimensions are available. OSM uses these
dimensions to cover all the different perspectives of the system. The navigation between these views is based on the navigation used in CAD tools. Figure 3.3 shows the basic idea of this principle. Each cube consists of several different cells represented by the different sides of the cube, from which the system can be viewed. Each cell represents a different dimension of the system. This differs from the normal tree-based structures and prevents the developer from having to learn the organizational structures of the development environment. The different dimensions can be selected by selecting a cell within the cube, that represents a view.

![Figure 3.3: Dimension-based navigation principle [19].](image)

In case the view needs a specific representation, this representation is automatically loaded in the environment. A structural view uses, for example, an UML-based environment, while the implementation view needs to show the source code in a specific language. In order to realize this, the environment supports different kinds of representation within the main environment, in this case the Eclipse IDE.

Figure 3.4 shows an implementation of the OSM methodology with all the seven dimensions on the right side of the picture. In Figure 3.4, the structural view of the component `TravelBookingSystem` is shown, using the `specification` dimension together with the `Service` granularity. Furthermore, the PIM abstraction, the latest version and all operations available are displayed. The OSM tool implementation is known as nAOMi, which is discussed in more detail in Section 5.2.

### On-the-fly View Generation

One of the most important features of OSM is the on-the-fly generation of views. Since all the information is present in the SUM, all information should be quickly and easily accessible. By using predefined transformations for the on-the-fly generation of views, it is no longer necessary for the developer to work on these transformations. Figure 3.5 illustrates this principle. The SUM contains all the information of the views, and the views can then be generated on-demand, whenever necessary. In this example, four different views are shown, however more views are possible.

The transformations between the views and the SUM are not always easy to create, however it requires less time than all the consistency checks that have to be developed in the traditional view-based approach. There are two options for the transformations, namely one-way and two-way transformations. When a view is read-only, only a transformation from the SUM to the view should be provided. However, when the developer wants to edit the view and submit it to the system, a transformation that merges the view with the SUM is also necessary. The latter is far more difficult than the former, since the transformation needs to
check whether the changes are consistent with the other parts of the model. Transformations in nAOMi can be described using the ATLAS Transformation Language ( ATL) [24].

Multi-level SUM-based modelling is a modelling methodology based on both SUM-based modelling as well as on multi-level modelling. The idea was introduced and partially explored by Atkinson et al. [10]. The multi-level SUM is based on the OSM architecture as well as on OCA. By combining these two, the benefits
of both languages can be used to create a powerful modelling environment.

Using this methodology, the SUM and most of the views are described using the multi-level syntax of LML. Section 4.1 shows that OSM uses the nAOMi implementation to model SUM-based models. Atkinson et al. extended this tool by allowing it to support modelling with multi-level models. All the different dimensions can now be views using the Melanee tool, which is discussed in more detail in Chapter 5. Since everything is based on multi-level concepts, the KobrA foundation had to be modelled in LML, such that it can be used to instantiate models on the O1 level. To give an example of the multi-level views, Figure 3.6 shows a behavioural view of a model. Using the graphical DSL capabilities of Melanee, the representation of the views can be visualized in any customized DSL. On the O0 level a part of the KobrA metamodel that is used for the behavioural aspects of the system is shown.

![Figure 3.6: A multi-level behavioural view, instantiated from a part of the KobrA metamodel [10].](image)

### 3.3.1 ATL Adapter

Since none of the transformation languages that are currently available supported multi-level transformations, an additional adapter had been developed [9] [11]. The adapter is an extension to the ATL transformation language, which is the most popular metamodel-based model transformation language available nowadays. The adapter has been developed in such a way that it would be easy for new developers to understand the extensions. Figure 3.7 illustrates how the adapter has been implemented. The adapter has been built on the Regular ATL Virtual Machine, and consists of three different components:

- **ASMPMModel** This component extends the default implementation of ASMEMFModel by loading ontological model elements as metamodel types. By doing this, users can define rules based on the ontological or linguistic elements of the multi-level model.
3.3. MULTI-LEVEL SUM-BASED MODELLING

3.3.2 Multi-level transformations

Although the multi-level SUM is based on OCA and implements multi-level transformations, transformations between two-level and multi-level models are still necessary in order to support the compatibility with MOF-based models, such as UML models. This means that three different kinds of transformations should be supported:

1. **Multi-level to multi-level transformations**
   These transformations are only used to transform models based on OCA, and are therefore not that hard to accomplish. Both models support multiple ontological levels and are thus rather easy to transform.

2. **Two-level to multi-level transformations**
   These transformations can be used, for example, to transform UML models to LML models. Because the multi-level models can also have two levels, these transformations are not that hard either. The first level can be transformed to the O0 level, whereas the instances can be transformed to the O1 level.

3. **Multi-level to two-level transformations**
   Multi-level to two-level transformations are used, for example, to transform an LML model to a UML model. These transformations are more difficult since the target model only supports two levels, whereas the source model could have more than two levels. In order to cope with this problem it should be possible to specify which ontological level should be transformed. Figure 3.8 gives an example of how the multi-level model (a) is transformed to a two-level model (b) and a multi-level model (c) when specifying a specific ontological level. For specifying this level the potency numbers

---

3.3. MULTI-LEVEL SUM-BASED MODELLING

- **AtlPLMModelHandler** This component delegates function calls to the multi-level implementations of the ASMModel and ASMModelElement.

- **ASMPLMModelElement** This component enables the accessibility to the ontological and linguistic elements and attributes. The default implementation of operations has been overridden to enable the reading and writing of the ontological and linguistic features, when defined on the ontological level. The linguistic requests are handled by the default ASMEMFModelElement implementation.

---

Figure 3.7: Architecture of the multi-level ATL adapter.
are used. When we specify potency "1", the next ontological level is transformed, for potency "2" the one after that, and so on. In case the number of levels is unknown, it is possible to specify potency "*", which always picks the last ontological level. Besides specifying an ontological level, it is also possible to specify a range of levels. This is especially useful for multi-level to multi-level transformations.

![Diagram](image)

Figure 3.8: Transformation without explicitly stating * potency defined on component [11].

Another feature added by the ATL adapter is the possibility to specify whether to target the ontological or the linguistic part of the element. This is done by adding ",l," for access the linguistic part and ",o," to access the ontological part. Besides this, it is also possible to distinguish between the different ontological levels. This is done by using a pattern like $PLM!O0\cdot Component$. In this case the Component on the ontological level O0 of the PLM model is accessed. Figure 3.1 illustrates part of a multi-level aware transformation using ATL.

```
1 rule Component2Class {
2   from
3     s : PLM\!O0\cdot Component
4   to
5      t : UML\!Class (
6       name <- s\_l\cdot name )
7   do {
8      thisModule\cdot umlModel\cdot packagedElement <- thisModule\cdot umlModel\cdot packagedElement ->
9         append(t); 
10    }
11}
```

Listing 3.1: Multi-level ATL code

### 3.4 Conclusion

In this chapter we discussed the concepts of view-based modelling and SUM-based modelling. Starting with the view-based modelling approach we discussed the current state and standardization of these concepts in ISO/IEC/IEEE 42010, using the conceptual model of Figure 3.1 to clarify the relations. Besides, we discussed the difficulties that arise with the consistency between the different views, and the techniques that are used to keep this consistency. Furthermore we discussed the SUM-based modelling concepts, starting with the OSM approach, which laid the foundation for many SUM-based techniques. In the end we discussed the idea of combining SUM-based modelling and multi-level modelling. Here we discussed the changes that had to be made to the model transformations in order to support multi-level modelling, which has been achieved by extending the ATL adapter. With respect to our design requirements we concluded that the current state-of-the-art of SUM-based modelling offers enough ground to implement our intended nAOMi extension.
Chapter 4

Available tool support

In this chapter we discuss the tools that we have used in this project. These tools have been chosen based on their ability to support multi-level modelling or SUM-based modelling. The general functionality and goals of these tools are discussed, as well as the current state of development and usability. In Section 4.1 we discuss the SUM-based modelling tool nAOMi. nAOMi has a multi-level version that is used for the implementation of our additional view. In Section 4.2 we discuss the multi-level modelling tool Melanee, which enables the design and representation of multi-level models. Both these tools are developed as Eclipse plug-in projects, which allows nAOMi to use Melanee to visualize the generated views.

4.1 nAOMi

Naomi is a SUM-based modelling tool that has initially been developed for the OSM SUM-based software modelling technique [34]. At the moment the tool is being extended to support multi-level modelling, based on the ideas proposed by Atkinson et al. [5]. Therefore, there are currently two versions available, namely the OMS-based tool, which is built for the Indigo version of Eclipse and uses uml2tools to visualize the views. The multi-level version is still under development, and currently uses the Luna version of Eclipse in combination with the multi-level tool Melanee to show the views. Melanee is an Eclipse plug-in which supports multi-level model design. The OSM version of nAOMi is available as a repository at http://www.Eclipselabs.com/naomi/. The multi-level version is available on request at the software engineering group of the University of Mannheim, where the tool is being developed. Because we are mainly interested in the multi-level version of nAOMi, we focus on this version.

4.1.1 Structure and functionality

Since nAOMi was built to support OSM SUM-based modelling, it adheres to the OSM principles. nAOMi uses a dimension based navigation system to show all the different available views. This dimension navigation system consists of the following dimensions:
• **Version**  
The version indicates which version of the system is shown. It is possible to switch back to older versions of the system.

• **Abstraction**  
The abstraction indicates whether the system is shown as a PIM (Platform-Independent Model) or as a PSM (Platform-Specific Model). The PIM is visualized in the standard concrete syntax, whereas the PSM is visualized in a DSL representation.

• **Component**  
The component indicates which component of the system is shown. By selecting a different component, different parts of the system can be shown.

• **Encapsulation**  
The encapsulation indicates whether to show the specification (black box) or the realization (white box) of the system.

• **Projection**  
The projection indicates which view of the system should be shown. This can be, for example, a structural view.

• **Granularity**  
The granularity indicates whether to show the services or the types of a system.

• **Operation**  
The operation allows us to select a certain operation of the selected component.

For all these dimensions there is at least one possible selection. Abstraction, encapsulation and granularity provide two options, whereas the other dimensions can provide one or more options. The component dimension usually provides most options, as this dimension enables to navigate through all the components present in the system. Depending on which options are selected, a view of the system can be generated. The type of the view is specified by the projection and is currently fixed at a structural view.

All the views are based on the Kobra-oriented SUM metamodel, shown in Figure 4.1. This metamodel, represented in LML, defines all the concepts used in Kobra to create the structural or behavioural views of a system.

![Figure 4.1: Kobra-oriented SUM metamodel [10].](image)
Views

The metamodel supports the structural and the behavioural view. Figure 4.1 shows that we can divide the Kobra metamodel into two parts: one consisting of the Component element with the four Associations, and another consisting of the elements State with its sub-elements Start and Stop, and Transaction. The former part is used to represent the structural view, and the latter part is used to represent the behavioural view. All the views are shown by using the Melanee plug-in. This plug-in allows the visualization of LML diagrams and allows these LML models to be edited.

Structural view The structural view is represented in LML. All elements created for the structural view are instantiated from Component, and are connected with each other using one (or more) of the Associations. The potency values of the Component and Associations determine that Component, Acquires and Creates can be instantiated over two levels, whereas Contains and Imports can only be instantiated once. This means that the only associations available for relations between runtime elements are Acquires and Creates, and that Contains and Imports are only used for the relation between the direct instances of the Component.

Behavioural view The behavioural view represents the state/transition diagram of the model. This view is currently not integrated within the tool, but is expected to be added in the future. The elements are instantiated from the State and connected with each other by Transactions. All the behavioural models have at least one Start state and one End state, to start and terminate the behaviours. The behavioural view is always designed at a single level since these diagrams are type abstractions, and therefore they can not be instantiated. This view is also more likely to be shown in a DSL representation than in the standard LML representation.

Transformations

In order to generate these views, nAOMi uses ATL as transformation language. Since ATL has been originally developed to support the native transformations, an adapter has been developed that implements the support for multi-level transformations [11]. This ATL adapter is used by nAOMi, such that both the ontological and the linguistic sides of the components can be transformed. Although this ATL adapter supports multi-level to multi-level, multi-level to two-level, and two-level to multi-level transformations, only the first is currently used in nAOMi. This means that the SUM and all its views are multi-level models. nAOMi supports two types of transformations, namely generation transformations, and merge transformations. Generation transformations are used to generate views from the SUM model, whereas merge transformations are used to merge the edited views with the SUM model. Merging a view with the SUM is done in two steps, first the view is transformed to a temporary SUM model, after which this model is merged with the old SUM model. During the last transformation the components of the temporary SUM model are compared with the components of the old SUM model, and are added, removed or updated.
4.1.2 Current state of development

Due to the early stage of development of nAOMi, there is still limited documentation available, which makes it sometimes rather hard to understand the structure and functionality of the tool. This also means that not all the functionality has been implemented yet, and there are still some bugs that need to be solved. At the moment the multi-level version of nAOMi supports only the structural view (projection) and even this view is not completely implemented yet. In the future the tool will be extended such that the complete view is supported, as well as the complete behavioural view. Furthermore, the version is always set to latest, the abstraction is set to PIM, the encapsulation to specification, the granularity to type and the operation to all owned. This means that currently only the selection of different components is supported. Since only one projection is partly supported, there is also only one generation and merge transformation available. The transformation that merges the new SUM with the old SUM is already completely implemented.

The tool has been developed in a modular way, which means that it can be extended by adding other modules to the program without affecting the original code. nAOMi is being developed as an Eclipse plug-in project, which uses extension points to create these modules. These extension points are currently available and contain variables necessary to link models and transformations.

4.2 Melanee

Melanee is a tool that has been built to support multi-level modelling. Melanee stands for Multi-lEvel modeleLing And oNtology Engineering Environment [33]. Melanee is based on OCA and therefore it uses PLM as abstract syntax and LML as concrete syntax. It supports multi-level modelling as well as all the other functionality that is implemented in the PLM, like the support for different modelling modes and reasoning services. Melanee is implemented as an Eclipse plug-in and the PLM is defined as an Ecore metamodel. This enables the support for the basic modelling features offered by EMF. In Melanee, LML is used for the graphical representation of models. The current available version is Melanee 2.0 and is available to download at http://www.melanee.org or at https://github.com/deep-modelling (Github repository). Melanee is also used as the model visualization in nAOMi, which means that all the functionality of Melanee is available in nAOMi.

4.2.1 Structure and functionality

Melanee consists of one core and five additional modules, all developed to support different aspects of modelling. The core model supports the main capabilities, like the generation of PLM content and the visualization of this content in LML. The five additional modules are described below. In addition to the core and the additional modules, third-party developers can also develop or propose extensions to Melanee.
4.2. MELANEE

Graphical domain-specific languages

Domain-Specific Languages (DSL) often prescribe the use of different notations for the elements and relationships between these elements. Therefore, Melanee supports not only the default multi-level representation, but also allows the user to create domain-specific renderings for each element or relationship. Custom representations are defined by using a tree-based editor based on the concepts of common GUI frameworks, like Standard Widget Toolkit(SWT) and Graphical Modelling Framework (GMF). This tree-based editor offers some standard visualizations, like lines, circles or rectangles, which can be configured accordingly, but also allows users to define complete new shapes by setting x/y-points. After the customization has been completed, the user can configure each element individually, by defining whether it should be visualized using the DSL or the default rendering. This enables symbiotic language support [7], which is the support of multiple visual representations within one model.

Textual Domain-specific languages

DSLs can be rendered with graphic elements, but also with textual elements. Therefore, Melanee does not only support graphical customization of language elements, but it also allows a textual representation. The implementation of textual DSLs is similar to that of graphical DSLs, and therefore it also allows symbiotic language support, by using elements with a textual representation and elements with a graphical representation simultaneously.

Reasoning services

One of the design goals of OCA was the support of different modelling modes. Traditionally, tools support either constructive modelling for the software engineering community or exploratory modelling for the knowledge engineering community. OCA supports both these modelling techniques and therefore it also enables reasoning services for both modes. Melanee, which is based on OCA, uses these reasoning services for two different purposes: ontology validation, which basically means that the model is considered as an ontology, and can be validated for correctness, and ontology population, which enables features like automatically defining instances that instantiate types, or identifying inheritance relationships between elements based on its characteristics.

Transformations

Since Melanee is based on multi-level modelling, and the currently available transformation languages are based on the traditional metamodelling principles, Melanee uses the ATL adapter developed to support multi-level transformations [11]. This adapter enables models created with Melanee to be transformed to and from, either two-level or multi-level models. By supporting also the transformations to and from two-level models, interoperability between Melanee and tools based on traditional modelling has been made possible.
Emendation

In order to make Melanee more user-friendly, it uses automated emendation, which ensures that changes made by the user directly affect the other elements connected to this change. In this way, the user does not have to change all these elements manually. Melanee keeps track of all the changes being made, and in case multiple elements are affected by the change, Melanee suggests to the user additional changes for the other elements. In this way, the user is aware of all the changes being made to the model, which reduces the chance of introducing inconsistencies.

4.2.2 Current state of development

Melanee 2.0 is currently still under development, but is already capable of supporting multi-level model design and transformations. Although it still lacks some documentation, there is a simple tutorial that helps people to get started with model design. The UI offers a clear overview of all the functionality and available components. Melanee offers an emendation service that notifies the user when certain parts of the model should be changed to prevent inconsistencies, which can then be done automatically or manually. Furthermore, there is graphical and textual DSL editor that allows customized model visualizations. All this functionality fosters the simplicity to design multi-level models. A downside of Melanee, especially in combination with the generation of views by nAOMi, is the layout and the alignment of the different components. A generated view often has overlapping components and connections, which creates a messy representation. There are some simple functions to align the components, but this could be improved, especially with nAOMi’s view generation. Additionally, the DSL editor could be extended with some more options regarding the connections, to improve the support for DSL representations.

Like nAOMi, Melanee 2.0 is being developed as an Eclipse plug-in and is therefore built in a modular way, which fosters the extension of the tool by third parties.

4.3 Conclusion

In this chapter the tools nAOMi and Melanee have been discussed. nAOMi focusses on SUM-based modelling whereas Melanee is mainly used for multi-level model design and transformations. The structure and functionality of both tools are explained and we observed that they are tightly connected to each other. Both tools are built as Eclipse plug-ins, which foster their modular extensions. Furthermore, nAOMi uses Melanee for the representation of the generated views, which shows the interoperability between these tools. With respect to their completeness, Melanee is certainly ahead of nAOMi. Melanee can be used for model design as well as transformations, and all necessary functionality for this has been implemented. nAOMi however, only supports a small part of the intended functionality and still requires a lot of added features in order to completely support SUM-based modelling.
Chapter 5

Context view design

In this chapter we discuss the development of the additional view and the extension of the structural view. We start with selecting a proper additional view. In Section 5.1 we discuss criteria to which the additional view should comply, and we discuss why we chose the context view in stead of the sequence view. In Section 5.2 we discuss the implementation of the context view. We start by designing the metamodel for the context view. From this metamodel an instance model is created to illustrate the possible instances. In order to represent the view in its intended visualization, a DSL representation has been designed using the custom DSL visualizing tool of Melanee. Finally we discuss the extension to the SUM metamodel in order to enable the utilization of the view.

Besides introducing an additional view, we also extended the structural view. Since the complete metamodel for this view is already present in the SUM metamodel, we only had to update the metamodel used for the view representation. In Section 5.3 we highlight the differences between the current and the extended implementation of the structural view.

5.1 View selection

As discussed in Chapter 3, views are an important aspect of SUM-based modelling. Each view represents a different aspect of the system, and often focuses on specific stakeholders. nAOMi addresses these different points of view with the use of dimensions, as described in Section 4.1. Although every different selective combination of these dimensions can be considered as a view according to the ISO/IEC 42010:2011 [32]. We use the term view specifically to identify a selected projection. A projection in nAOMi specifies which view is selected, e.g., a structural view or a context view.

In order to extend nAOMi with an additional view, we sought for a view that would enhance the usability and the grasp of the tool. Numerous views are available, which are all useful for their own purposes, thus some choices had to be made. In order to find a view that enhances the usability, we created several criteria to which it should comply.

1. The additional view should add new possibilities to the SUM-based modelling with nAOMi. Since a
lot of views are closely related to each other, we have to find a view that should not look too much like the partly implemented structural view or like a behavioural view. The latter because it is planned to be added by the nAOMi developers.

2. The additional view has to be added to the SUM metamodel and thus should be connected to any of the entities of the structural view. This means there should be a logical connection between one of the entities of the additional view and the structural view.

3. The additional view should be generated and represented in a DSL customized visualization. This visualization is designed using the DSL editor of Melanee, and is thus limited to its capabilities.

After the first criterion we chose to have a closer look at two views, the sequence diagram and the context diagram. Both views have unique abilities. The former focuses on the processes between components and the order in which they occur, whereas the latter focuses on the system boundaries and the interaction with its stakeholders. In order to check the other criteria and to see which view would be most suitable, we created a SUM metamodel extension concept and looked into the visualization of the DSL representations for both views.

5.1.1 Sequence diagram

A sequence diagram is a diagram that focuses on the communication between different components of a system. Each component is connected to a process which visualizes the duration. During this process messages are sent to, or received from, other processes. Figure 5.2 shows an example of a sequence diagram. In this example there are three components that all have two processes. Between these processes exists two types of messages, namely synchronous (normal arrow) and response (dashed arrow) messages. Besides these messages, sequence diagrams also support asynchronous and initialisation messages. The messages are ordered on the time of occurrence, from top to bottom. This makes the sequence diagram a great tool to visualize the inner system communication.

In order to validate criterion 2, we created a metamodel extension for the sequence diagram. For this metamodel we had to model the component, process and the four types of messages. Furthermore, we had to connect this metamodel with the existing SUM metamodel. Since the structural view already contains an entity component, we used this entity to connect to the process. This ensured the logical connection between the sequence metamodel and the SUM metamodel, as desired from criterion 2. The process is able to contain messages connected to itself. The four supported messages are modelled as sub-connections of message. Furthermore, entities have several attributes, like the the startTime and endTime to indicate the duration of the process, and a name, time and condition to indicate the name, time and condition of the message that is sent. Figure 5.1 shows the metamodel extension for the sequence view.

Since we met the second criterion, we could look into the DSL visualization of criterion 3. Figure 5.2 shows a simple sequence diagram in the intended visualization. The goal was to design a DSL, such that it looks like the visualization of Figure 5.2. In order to design this DSL we used the visualization editor of Melanee. Because the visualization editor contains some limited functionality, we were unable to create a perfect DSL representation. However, we managed to create a similar representation, as shown in Figure 5.3.

Although this visualization would be acceptable and is clear enough for the user to interpret it as a sequence
5.1. VIEW SELECTION

Figure 5.1: SUM metamodel extension concept for a sequence diagram

diagram, it would not be impossible to generate the views like this. The main problem is the size of the rectangles that represent the processes, and the location of the messages. The size of the process depends on its duration, as defined by its startTime and endTime attributes, whereas the location of the messages are defined by the time attribute. After some tests, we concluded that it is impossible to specify the size and location based on attributes. Besides this, there are connections between every process and its component, which could not be hidden or placed in the background. Altogether, this view would be quite messy for visualization. This means that it does not meet third criterion, since it does not allow the view to be generated in a correct DSL visualization.

Figure 5.3: DSL representation for a sequence diagram
5.1.2 Context diagram

A context diagram is a relatively simple diagram that is used to show the boundaries of a system (or subsystem) and its environment, by showing the interaction with all actors. Actors can be human entities as well as other software systems. Context diagrams are high level diagrams that hide system details, but focus only on the system environment and the communication relations between the system and its actors. Every context diagram contains only one system and can contain one to many actors. Figure 5.4 shows an example of a simple context diagram.

This context diagram shows a simple example of a file report system that interacts with two different actors: a database system, which means this database system is not part of the system under development but rather an external entity, and the user who analyses the reports and commits chances when necessary. This actor is a human entity which can also be represented by a stick figure to emphasize the human interaction. The lines between the system and the actors are called relationships, and represent their interactions. The arrowhead of this line indicates the direction of this communication.

Although context diagrams can be used at any time throughout the life cycle of a system, they are particularly useful for identifying the requirements of the system. Due to the clear indication of the boundaries of the system and the communications from and to the system, one can easily identify which requirements should or should not be part of the requirements specification. Furthermore, the relationships between the system
and the actors help specify which functionality should be included, in order to support this relation. For example, in Figure 5.4 the system needs to support functionality to send reports to the users, in order to support the send report relation.

![Figure 5.5: Context metamodel concept](image)

In order to check criterion 2, we will design a metamodel for the context diagram. We can distinguish two different entities, namely system and actor. Between these entities there are relationship connections to model the interactions. Figure 5.5 shows the metamodel. Criterion 2 asks for a logical connection between the context metamodel and the structure metamodel. We chose to connect system with component of the structural view, since a system consists of multiple components. Because the design and connection of the metamodel did not create any problems, we conclude that criterion 2 has been met.

![Figure 5.6: DSL representation for a context diagram](image)

The third criterion asks for the context view to be generated and represented in a customized DSL. Figure 5.6 shows the same system as in Figure 5.4, but now represented in the customized DSL. This DSL has been designed using the visualization editor of Melanee. The only problem we encountered during the design of this DSL were the visualization of the lines. Due to the functional restrictions of the visualization editor, we were unable to created curved lines. Since the shapes of the entities do not depend on any attributes, the generation of this DSL representation does not create any difficulties. Although the visualization is not exactly the same as in 5.4, we think that it is sufficient enough to meet criterion 3.

### 5.1.3 Selection

After testing the sequence diagram and the context diagram for criteria 2 and 3, we found that the design of the metamodel was no problem. However, designing a customized DSL representation that can be generated was a slightly harder exercise. For both diagrams we had some problems to design the correct representation. Because we are unable to generate the sequence view in the designed DSL representation, and the context view did not have this problem, the context view is a good option to extend nAOMi. Although the
representation of the view is not the most important aspect of our extension, it would be better to visualize the view correctly.

Before making a final decision we look into the limitations of the context diagram. The main limitations of context diagrams are: (1) they do not allow the representation of internal system functionality, (2) they do not allow the representation of timing or order of the interaction between the system and the actors, and (3) Limited support for systems has different working modes. An example of the latter is a system that interacts with only one actor in sleeping mode, but interacts with multiple actors in operational mode. Because this view will be used in a SUM-based environment, one should be able to switch between different views. This solves the limitation concerning internal system functionality, as this can be shown by the structural view. The limitation concerning time and order of the interactions between the system and actors could be solved by implementing a sequence diagram as an additional view. Thus by implementing a context view in a SUM-based environment such as nAOMi, the two limitations can be compensated by other views. Even with the remaining limitation, it would still be possible to model the complete system with all actors. However, it is impossible to distinct actors that use different working modes.

Since the context model meets all the criteria and does not have severe limitations, we think that implementing a context view would be a useful addition to nAOMi.

5.2 Context view implementation

In this section we discuss the process of designing models for an additional view. For each view addition to nAOMi, there are three steps that should be performed. These steps only concern the modelling part of the implementation. The transformations of the additional view are discussed in Chapter 6. The following steps have been defined:

1. Design a metamodel for the additional view.
2. Design a customized DSL representation for the additional view.
3. Extend the SUM metamodel with the metamodel of the additional view.

We followed these steps during the implementation of our addition view. Each step will be described in detail, providing figures to show the results.

5.2.1 Metamodel design

During our view selection process we created a metamodel for a context view. However, during the development of our extension we noticed that this metamodel was not sufficient to model all the capabilities of the context view. Furthermore, we chose to support not only one system, but also multiple sub-systems. This allowed us to create more detailed models. Although some entities and connections have been added, the metamodel was still rather easy to define.

First of all we had to define the entities that represents the system and subSystem, which should contain
an attribute name to specify its name. Each system consists of several sub-systems, which are connected to each other by a consistOf relation. Besides, we defined an entity that specifies the actor, which also contains an attribute name. In order to specify the type of the actor, e.g., a human or another system, an attribute type has been added. Between the subSystem and actor there are two different connections: relationshipFrom, to specify the relation that originates from the subSystem and points to the actor, and relationshipTo, to specify the relations going to the subSystem and coming from the actor. Furthermore, an internal relationship relationshipInternal specifies relations among the subsystems themselves, and an external relationship relationshipExternal specifies the relations between actors. These relationship connections contain an attribute description, which specifies the description of the relationship. The complete metamodel of a context view is shown in Figure 5.7.

Model instances

The metamodel in Figure 5.7 can be used to instantiate models. Figure 5.8 shows an example of a model on level O1, which uses the instances of the O0 metamodel. In this model we see one system, MainSystem, connected to two sub-systems, sub1 and sub2, by the consistOf connection. Furthermore, there are two actors, Customer and Admin. The sub-systems and actors are all connected to each other by the different relationships. Because this model is represented in the default LML visualization of Melanee, the model does not look much like a context diagram, such as that of Figure 5.4.

5.2.2 DSL implementation

In order to achieve the same representation, we used the visualization editor of Melanee. This editor has several pre-made shapes that can be enhanced by using specific properties. For every entity or connection in the metamodel, a graphical DSL visualizer has been added. This visualizer determines how the instances of this entity or connection should look like. The visualization editor works with a tree-based structure, as
shown in Figure 5.9, which shows the DSL representation for the system entity.

During our view selection procedure we created a DSL customization. Although we added some new entities and connection, the basics of this customized DSL stayed the same. The system and sub-system entities should be represented as a circle, with their name shown in the center. This is achieved by applying the visualization of Figure 5.9. This visualization consists of several specifications, such as the circle and the label expression specifying the shape and the text respectively. The table layout and table layout information are used to determine the position of the label. The visualization of the actors and the relationships are defined in a similar way. By enabling the DSL representation option for every element at the instance level of the model, the visualization of the model, shown in Figure 5.10, is achieved.

The actors are shown as rectangles with the name specified in the center. Because we were unable to define a shape according to the value of an argument, the different types of the actors can not be displayed by different shapes. For example, human actors can not be represented by a stick figure, unless the other actors are also represented by the same figure. Therefore, we chose to visualize all actors as a rectangle. Relationships are
5.2. CONTEXT VIEW IMPLEMENTATION

Figure 5.9: DSL visualization editor for system entity

represented by an arrow, with the description of the relationship above it. Due to the graphical limitations regarding the lines between entities, these connections could not be shown as curved lines, but had to be displayed as straight angular lines. Although the visualization is not exactly as the original visualization of context diagrams, it is close enough because it allows one to distinguish the systems from the entities, and to recognize the relationships and their meaning.

Figure 5.10: DSL representation of the context model.

5.2.3 SUM metamodel extension

Figure 5.11 shows the original implementation of the SUM metamodel. This metamodel contains the entities necessary to support a structural view and a behavioural view, divided by the StateContainment connection. We extended this metamodel by adding our context metamodel (Figure 5.7) to the left side of the metamodel, by connecting subSystem to component. By connecting the subSystem to component we can indicate that the subSystem consists of one or multiple components, which is indeed the case. In order to implement this connection, we introduced an includes connection, to connect subSystem with component. This results into a new SUM metamodel that supports a structural view, behavioural view and our context view, as shown in Figure 5.12.

Although the context view does not visualize the components connected to the system, it is necessary to specify this connection in the SUM metamodel. Without this connection we would be unable to determine which sub-system should be shown when a certain component is selected. Furthermore, it joins the structural view with the context view, which unites the system as a whole.
Helper view

The helper view is specially developed to connect the `subSystems` with `components`. The view only contains the entities `subSystem` and `component`, together with the connection `includes`. Figure 5.13 illustrates how this view looks like. This view does not use a DSL implementation and always shows all available `subSystems` and `components`.

5.2.4 Alternative representations

Besides the view shown in Figure 5.10, alternative representations of the context view are supported. With the `Complete overview` option it is possible to show the complete system, instead of just one sub-system. Depending on the selection of the `encapsulation` either the complete view is shown, including the sub-systems, or only the system is shown, hiding the subsystems. The `encapsulation` provides two options, namely the `realization` (white box) and the `specification` (black box). The `realization` shows the system with all sub-systems and actors, as shown in Figure 5.10, whereas the `specification` only shows the system with the actors. Figure 5.14 shows an example of the same system with the `specification` option selected. The
5.3 Structural view extension

nAOMi currently only supports part of the structural view. One of our goals is to extend this view such that it is fully functional. By doing this together with introducing the context view, two functional views will be available to analyse and edit the system.

The original metamodel for the structural view is shown in Figure 5.15. This metamodel contains a subjectComponent, which is connected to a component by an acquires relation. The subjectComponent refers to the component that has been selected in the dimension explorer of nAOMi. All the components connected to the subjectComponent are modelled as normal components. Furthermore, the metamodel only supports one of the four possible association relationships. By using this metamodel for the structural view, the functionality of this view is limited, since three out of four associations can not be represented. Besides, the acquires association is modelled between a subjectComponent and a component, meaning it is not possible to create this connection between two components, although according to the SUM metamodel this connection should be allowed.
In order to solve this problem, the metamodel for the structural view needs to be replaced by the 'structural' part of the SUM metamodel, which is shown on the left side of stateContainment in Figure 5.11. This enables the complete utilization of all possibilities of the structural view, and models the view as it was originally intended.

5.4 Conclusion

In this chapter we discussed the modelling implementation of the additional view. We started with a view selection procedure were we tested a sequence diagram and a context diagram based on the defined criteria. During this process we found that generating the DSL representation for the sequence diagram was impossible with the current functionality of the visualization editor. Since the context diagram met all the criteria, we chose to extend nAOMi with a context view. The implementation of this view consists of three steps, which have all been discussed in detail. First we designed the context metamodel and showed how its instance model looks like. Since the instance model was still shown in the LML representation, we designed a customized DSL representation to show the context view in its intended representation. Next we extended the SUM metamodel with our context metamodel, to allow the view to be used in the SUM-based environment. We looked into the alternative representations that are especially useful to give a clear overview of the system. Finally we briefly discussed the extension of the structural view, which is now completely supported.
Chapter 6

Model transformations

This chapter discusses the transformations necessary to implement an additional view, as well as the transformations to extend the structural view. We start by introducing ATL, which is the transformation language used to write our transformations. After that we examine the transformation chain, which shows the order in which the different transformations are executed. Next we discuss the transformations in detail, by explaining some parts of the transformation code. Since most transformations have similar functionality not all transformations will be discussed in depth. In total, there are seven transformations, of which the complete code can all be found in Appendices A to G.

6.1 ATL

All transformations are written in the ATLAS transformation language (ATL) [24]. ATL is a popular transformation language, initially designed for MDE model transformations. Traditionally, it does not support multi-level model transformations. However, an ATL extension had been developed to support multi-level transformations, which is used by nAOMi. ATL is a hybrid language that uses both imperative as well as declarative programming, though most of the transformations consists of declarative statements. The mainly used concepts of ATL are rules, lazy rules, guards and helpers.

1. **Rules**: Rules are statements that match certain components of the source model to components of the target model. Listing 6.1 shows two rules, namely actor and relationshipFrom. Each rule consist of a from, to and optionally a do section. The from section defines which component of the source model should be transformed, and can contain possible guards to add requirements to the rule. The to section defines the target component, and contains the mapping of the source attributes to the target attributes. Finally, the do section is the imperative section, and is executed when the rule is finished. The do section is often used to perform certain statements that are hard to achieve in the declarative section.

2. **Lazy rules**: Lazy rules are similar to rules, but can be called from other places, whereas normal rules are only performed when matched components are found in the source file.
3. **Guards**: Guards are statements that can be added to the *from* section of a rule. Guards add extra requirements to the execution of the rule. For example, the rule *actor* in Listing 6.1 contains two guards at line 3. The actor is only transformed when either of these guards return true.

4. **Helper**: Helpers are used to define methods or attributes that can be called from other places in the transformation. Listing 6.1 contains several calls to helpers, all starting with *thisModule*. Please note that *thisModule* can also be used to call lazy rules. Listing 6.3 shows several helpers. Each helper starts with "helper def", followed by its name, type and value.

Besides these main concepts, ATL uses attributes, collections and other concepts that are not discussed in detail. During the explanation of the different transformation, we will explain these concepts if found necessary to understand the transformation.

### 6.2 Transformation chain

The transformations allow us to generate and merge the views designed in Chapter 5. Each view requires two transformations, one for the view generation and one to merge the view with the SUM model. Since nAOMi already supports a part of the structural view, a part of the structural transformation was already implemented. However, just like the view, the transformation is incomplete and needs to be extended in order to support the complete structural view capabilities. Besides the view transformations, a transformation that merges the old SUM model with the new SUM model is also necessary. This transformation is already provided with the current tool. Finally a small transformation called the helper transformation is used to connect the entities of the two views to each other.

![Figure 6.1: Visualization of the transformation chain](image)

Figure 6.1 shows the transformation chain, as used in nAOMi. Each arrow-shaped pentagon represents a transformation. These transformations use source models as input, and create target models as output. For example, transformation *T1* uses the models *parameter* and *SUM model* as input, and creates the *context* view.
view as output. This is the first step in the transformation chain. The next step is transformation $T_2$, which uses the context view as input, and produces the temporary SUM. Finally, transformation $T_7$ is executed, which transforms the input models temporary SUM and SUM model into the SUM model. For this last step the SUM model is synchronized with the temporary SUM, which results in the new SUM. After the final transformation the process can be repeated. All views use a similar transformation chain to the one described above. The only difference is that the first two transformations are slightly different. From Figure 6.1 we can define the following available transformations:

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>This is the generate context view transformation, discussed in Section 6.3.1. This transformation uses the SUM model and parameters as input models, and transforms them to the context view.</td>
</tr>
<tr>
<td>$T_2$</td>
<td>This is the merge context view transformation, discussed in Section 6.4.1. This transformation uses the context view as input model, and transforms it to the temporary SUM.</td>
</tr>
<tr>
<td>$T_3$</td>
<td>This is the generate structural view transformation, discussed in Section 6.3.2. This transformation uses the SUM model and parameters as input models, and transforms them to the structural view.</td>
</tr>
<tr>
<td>$T_4$</td>
<td>This is the merge structural view transformation, discussed in Section 6.4.2. This transformation uses the structural view as input model, and transforms it to the temporary SUM.</td>
</tr>
<tr>
<td>$T_5$</td>
<td>This is the generate helper view transformation, discussed in Section 6.3.3. This transformation uses the SUM model as input model, and transforms it to the helper view.</td>
</tr>
<tr>
<td>$T_6$</td>
<td>This is the merge helper view transformation, discussed in Section 6.4.3. This transformation uses the helper view as input model, and transforms it to the temporary SUM.</td>
</tr>
<tr>
<td>$T_7$</td>
<td>This is the merge SUM transformation, discussed in Section 6.5. This transformation uses the temporary SUM and SUM model as input models, and transforms them to the new SUM model. This is the last transformation in the transformation chain, and ensures that the SUM model is synchronized with the chances made in the views.</td>
</tr>
</tbody>
</table>

6.3 Generate view transformations

For each view available in the system, one generate view transformation should be defined. From our transformation chain we see that we need to define three generate view transformations, namely $T_1$, $T_3$, and $T_5$. The name of these transformations consist of (a part of) the view name, followed by $SUM2View$, and separated by an underscore. This results in the following three generate view transformations: context$_{SUM2View}$, struct$_{SUM2View}$ and helper$_{SUM2View}$, for the context, structural and helper view respectively. From now on we refer to these transformations by using these names.
6.3.1 Generate context view

The context_SUM2View transformation is used to generate the context view. In order to generate this view, the transformation uses the SUM model, which contains all the elements of the system, and the parameters model, which contains the current settings with regard to the selected dimensions. At this moment we use two parameters, one for the selected component, and one for the selected encapsulation. The former is stored in the helper black, which is a boolean that specifies whether the encapsulation black box is selected or not. The latter is stored in the helper subjectName, which is a string that contains the name of the selected component. These specific helpers are often used in guards or do-sections, and can be seen in lines 3, 17, and 27 of Listing 6.1. These helpers help us to determine how the view should be represented. It provides different options, such as the complete overview, black box and white box representations of the system. In this section we discuss some examples of the transformation. The complete context_SUM2View transformation can be found in Appendix B.

```java
rule Actor {
  from
  s : SUM!"O0. Actor" ((thisModule.act -> includes(s)) or (thisModule.subjectName = thisModule.all))
  to
  t : Context!"O0. Actor" (".xmlID" <- s."xmlID",
    name <- s.name,
    type <- s.type,
    ".name" <- s."name",
    ".potency" <- s."potency",
    ".feature" <- thisModule.createMethods(s))
}
rule RelationshipFrom {
  from
  s : SUM!"O0. RelationshipFrom" ((thisModule.relf -> includes(s)) or (thisModule.relfi -> includes(s)) or (thisModule.subjectName = thisModule.all))
  to
  t : Context!"O0. RelationshipFrom" (".xmlID" <- s."xmlID",
    ".name" <- s."name",
    description <- s.description,
    ".feature" <- thisModule.createMethods(s))
  do
    if (thisModule.encapsulation = thisModule.black)
      { t.".participation" <- (s.getAllParticipations() -> collect (e | thisModule.redirectParticipations(e)));
      }
    else
      { t.".participation" <- (s.getAllParticipations() -> collect (e | thisModule.createParticipations(e)));
      }
}
```

Listing 6.1: Context_SUM2View: Actor and RelationshipFrom rule

Since the metamodel of the context view is a part of the SUM metamodel, all the entities and connections are transformed to similar entities and connections in the context view. Listing 6.1 shows two transformation
6.3. GENERATE VIEW TRANSFORMATIONS

rules for important entities of the context view.

Rule *Actor* transforms the actor entity in the SUM model to an actor entity in the context model, as shown in lines 1 to 13 of Listing 6.1. Because these are ontological entities on the O0 level of the model, we need to specify this by using quotation marks to indicate the ontological side. Furthermore, we need to specify the level on which the entity occurs, in this case O0, followed by a dot and the name of the entity. In line 3 a guard has been specified that is used to check whether the matched *actor* is connected to the *subSystem* that includes the selected *component*, or whether the name of the selected component is equal to *complete overview*, which is used to show the complete system.

Once the element on the target model is created, its attributes are populated according to the transformation definition in lines 5 to 11. Since the models are multi-level models, and can make a distinction between the linguistic and the ontological sides, the prefix "L" has been added in order to call the linguistic side of the entity. Without this prefix, the ontological side of the entity is called by default. In line 11, the methods of the entity are transformed using the helper *createMethods*, that creates a sequence of methods. Since methods are of type *feature*, they can be mapped to feature of the target element.

Rule *relationshipFrom* is used to transform connections from *subSystem* to *actor*, and is defined in lines 15 to 36. The guards defined in line 17 are used to check whether the connection contains a participation with the target destination set to the selected *subSystem*, that includes the selected component. Furthermore, it checks whether the participation destination connects to an *actor* and to a *subSystem* that are both connected to the selected *subSystem*, or whether the selected component is equal to *complete overview*. The attributes and methods of the connection are transformed similarly to those of the rule *actor*. Rule *relationshipFrom* contains a do-section in lines 25 to 35 that set the participations of the connection. Depending on the encapsulation setting, the participations are either set to the original source *subSystem*, or redirected to the *system* entity. In order to redirect the participations, lazy rule *redirectParticipations* is used. This lazy rule checks whether the *complete overview* is selected, and sets the destination of the participations to the *system* entity, if this is the case.

In addition to the rules for *actors* and *relationshipFrom*, there are also rules to transform the *subSystem*, *system*, *relationshipTo*, *relationshipInternal* and *relationshipExternal* elements. These rules are similar to the rules described above and are omitted here.

Complete overview

Besides generating the context view for the selected component, it is also possible to generate the context view for the whole system. This is done by selecting the *Complete overview* that has specially been added to the list of components for this purpose. Listing 6.1 shows that each rule contains guards that check for the *complete overview* option. When this guard holds and the encapsulation is set to *white box*, all the entities and connections will be transformed. When the encapsulation is set to *black box*, all the *subSystems* are omitted, and all the relations between *actors* and *subSystems* are redirected to the *system* entity. This is achieved by the do-section in lines 25 to 35 of Listing 6.1.
6.3.2 Generate structural view

The `struct_SUM2View` transformation is used to generate the structural view. In order to generate this view, the transformation uses the SUM model, which contains all the elements of the system, and the parameters model, which contains the current settings with regard to the selected dimensions. Currently only the parameter that defines the selected component is used for the structural view. The value of this parameter is stored in the helper `subjectName`, which is a string that contains the name of the selected component. In this section we discuss some examples of the transformation. The complete `struct_SUM2View` transformation can be found in Appendix A.

```plaintext
rule Component {
from
  s : SUM! "O0. Component" ( (thisModule.com->includes(s)) or (s = thisModule.sub) ) or (thisModule.
  subjectName = thisModule.all ) )
t : STRUCT! "O0. Component" ( 
  "l_.xmiID" <= s_.l_.xmiID, 
  name <= s_.name, 
  "l_.name" <= s_.l_.name, 
  "l_.feature" <= thisModule.createMethods(s) )
}
rule Creates {
from
  s : SUM! "O0. Creates" ( (thisModule.cre->includes(s)) or (thisModule.subjectName = thisModule.
  all ) )
t : STRUCT! "O0. Creates" ( 
  "l_.xmiID" <= s_.l_.xmiID, 
  "l_.name" <= s_.l_.name, 
  "l_.feature" <= Sequence{thisModule.createMethods(s),thisModule.createAttributes(s)}, 
  "l_.participation" <= (s_.getAllParticipations() -> collect (e | thisModule.
  createParticipations(e)))
)
}
```

Listing 6.2: Struct_SUM2View: Component and Creates rule

The purpose of the `struct_SUM2View` transformation is similar to that of the `context_SUM2View` transformation, namely to transform a view according to a metamodel that is part of the SUM metamodel. Both transformations use the SUM model and the parameters model as input. Furthermore, both have to transform only those entities and connections that are connected to the selected component. Therefore, there are a lot of similarities between those transformations. Listing 6.2 shows two main rules from the `struct_SUM2View` transformation.

Rule `component` transforms the `component` element. This rule is performed in case the guards, specified in line 3 of Listing 6.2, hold. These guards check whether the component is either the subject component or a component that is connected to the subject component, or whether the subject component is equal to the Complete overview. In the latter case all components are transformed. Once the target element is created, the attributes are mapped according to the transformation definition in lines 5 to 9.

Rule `creates` transforms the connection `creates`, which is a connection between two components. Just like the guards of the `component` rule, the guards of the connection check whether it is connected to the `subject component`. Furthermore the `creates` connection contains features that are transformed with the use of two
6.3. GENERATE VIEW TRANSFORMATIONS

helpers, namely the `createMethods` and `createAttributes` (line 19). Both these helpers return a sequence, and are mapped to the `feature` of the target. Since both `methods` and `attributes` are of type `feature`, this is allowed. Besides the `creates` connection, three more connections can be established between components, namely `acquires`, `imports` and `contains`. The rules to transform these connections are similar to the `creates` rule, and are omitted here.

Complete overview

Similar to the context view, the structural view also includes the option to show the complete overview of the system. Unlike the context view, however, no special rules and helpers are required to show the right entities and connections. To show the complete overview of a systems, all the `components` with all the connections are shown in one view. In order to check whether the Complete overview has been selected, a guard has been added to each rule in the transformation, which checks if the name of the selected component is equal to Complete overview (lines 3 and 14, Listing 6.2).

6.3.3 Generate helper view

The `helper_SUM2View` transformation is used to generate the helper view. Like the context and structural view, it uses the SUM model as an input model. However, the helper view does not use the parameters model because it will always show all the entities specified in its metamodel. The helper view consists of structural entity `components`, the context entity `subSystems`, and the connection `includes`, and is used to link the context view with the structural view. Since the rules for these entities are similar to the `component` rule of Listing 6.2, we will not include any example code for this transformation. The only difference with the `component` rule of Listing 6.2 is that no guards are specified in `helper_SUM2View`. The complete `helper_SUM2View` transformation can be found in Appendix F.

6.3.4 Helpers

In the generate view transformations we use several helpers, e.g., to check whether a certain entity is connected with another entity. Listing 6.3 shows helpers that are used to determine whether the `actor` is connected to a `subSystem` that includes the selected component. In order to simplify this check, the helpers have been split into several smaller helpers. Listing 6.3 shows an example from the context view, but similar helpers are used in the structural view, concerning the structural entities and connection.

```plaintext
-- get all the actors connected to the subSystem.
helper def : act : OclAny = SUM!"O0. Actor"->allInstancesFrom('IN')->select (i | ((thisModule.relf
  ↪→ exists (j | (j.getParticipants() ↪→ exists (k | k = i)))) or (thisModule.relt ↪→
  exists (j | (j.getParticipants() ↪→ exists (k | k = i))))));

-- Get all the relationship connections that are connected to the system.
helper def : refl : OclAny = SUM!"O0. RelationshipFrom"->allInstancesFrom('IN')->select (i |
  thisModule.conn(i));
helper def : relt : OclAny = SUM!"O0. RelationshipTo" -> allInstancesFrom('IN') -> select (i |
  thisModule.conn(i));
```
Helper act selects all the actors that have either a relationshipFrom or a relationshipTo. Helpers relf and relt use helpers conn and part to check whether these relationships are connected to a subSystem. Helper syss checks whether this subSystem is connected with the selected component, using helpers inc, con, and par. In the end, only the actors for which all of these helpers return true, are put into a sequence. In order to check whether the current actor is included in this sequence, one can use the includes operation on these helpers. An example of this can be seen in line 3 of Listing 6.1.

Listing 6.4 shows helper createMethods which is often used in the generate view transformation, to map the methods to the features of the target element. A similar helper called createAttributes has been defined to map attributes. Helper createMethods creates a sequence of all the methods that are part of the given entity. For this entity we iterate through the features, and check for every feature whether it conforms to a method type. If this holds, the unique lazy rule createMethod(i) is called, which creates the method and adds it to the sequence. A unique lazy rule makes sure the method is created only once, even in case there are multiple occurrences of this method in the sequence.

6.4 Merge transformations

Besides the generate view transformations, every view requires a merge view transformation. The merge view transformations are the first step of synchronizing the edited view with the SUM. The second step is the merge_SUM transformation, which is discussed in Section 6.5. From our transformation chain (Figure 6.1) we see that three merge view transformations are required, namely T2, T4, and T6. The merge view transformations are called context_View2SUM for the context view, struct_View2SUM for the structural view, and helper_VIEW2SUM for the helper view. In this section we will discuss these views in detail.
6.4. MERGE TRANSFORMATIONS

6.4.1 Merge context view

The context View2SUM transformation is used to transform the view that has previously been generated, and optionally edited, to a temporary SUM model. In order to transform this view, all the entities and connections of the context view are transformed to similar entities and connections that adhere to the SUM metamodel. Listing 6.5 shows two main rules from the context View2SUM transformation. The complete context View2SUM transformation can be found in Appendix D.

```plaintext
rule Actor {
    from s : PLM!!O0. Actor
    to t : PLM2!!O0. Actor (",
    _l_.xmlID   <- s._l_.xmlID
    name   <- s.name
    type   <- s.type
    _l_.name <- s._l_.name
    _l_.potency   <- s._l_.potency
    _l_.feature   <- thisModule.createMethods(s)
}

rule RelationshipFrom {
    from s : PLM!!O0. RelationshipFrom (not(thisModule.relf -> includes(s)))
    to t : PLM2!!O0. RelationshipFrom (",
    _l_.xmlID   <- s._l_.xmlID
    _l_.name   <- s._l_.name
    description <- s.description
    _l_.participation <- (s.getAllParticipations() -> collect (e | thisModule.
                        createParticipations(e)))-,
    _l_.feature   <- thisModule.createMethods(s)
}
```

Listing 6.5: Context View2SUM: Actor and RelationshipFrom rule

Since all the entities and connections of the view are transformed to the temporary SUM, there are almost no guards necessary for the rules. This means that the overall transformation is easier compared to that of the context SUM2View transformation. All the helpers from Listing 6.3 are no longer required. The only guard that is used in this transformation is the guard shown in line 18 of Listing 6.3, in the rule RelationshipFrom. This guard makes sure that the relationshipFrom is only transformed if it is not connected to a system entity. This is necessary to prevent the relations to be permanently redirected from the original subSystem to the system, when the black box and complete overview options were set upon generating the view.

Listing 6.5 contains two rules, namely actor and relationshipFrom. Rule actor works the same as the actor rule from context SUM2View, except it doesn’t contain any guards. Rule relationshipFrom also works similar to the relationshipFrom rule in context SUM2View, however, the participations are handled differently. This relationshipFrom rule does not contain a do-section, and always transforms the participations using a lazy rule, called createParticipations. This lazy rule is called at line 24, and returns a participation created from the given source participation. Finally the collection of these participations are mapped to the target participation. Besides the actor and relationshipFrom rules shown in Listing 6.5, the transformation contains similar rules to transform subSystem, system, relationshipTo, relationshipInternal and relationshipExternal.
6.4.2 Merge structural view

The *struct View2SUM* transformation is used to transform the structural view to the temporary SUM model. Similar to the *context View2SUM* transformation, the *struct View2SUM* transformation has to transform all its entities and connections. Since the metamodel of the structural view is a part of the SUM metamodel, the source and the target of the entities are similar. Listing 6.6 contains two rules from the *struct View2SUM* transformation. The complete *struct View2SUM* transformation can be found in Appendix C.

```plaintext
rule Component {
    from s : PLM!"O0. Component"
    to t : PLM2!"O0. Component"
    "l.xmiID" <- s.l.xmiID,
    name <- s.l.name,
    "l.name" <- s.l.name,
    "l.feature" <- thisModule.createMethods(s)
}

rule Creates {
    from s : PLM!"O0. Creates"
    to t : PLM2!"O0. Creates"
    "l.xmiID" <- s.l.xmiID,
    "l.name" <- s.l.name,
    "l.feature" <- Sequence{thisModule.createMethods(s), thisModule.createAttributes(s)},
    "l.participation" <- (s.getAllParticipations() -> collect e | thisModule.createParticipations(e))
}
```

Listing 6.6: Struct_View2SUM: Component and Creates rule

The merge transformation for the structural view does not require any guards to the rules. Therefore, all the entities of the view are transformed. Listing 6.6 contains the rules *component* and *creates*. Both rules contain the same functionality as we have seen before in the *struct_SUM2View* transformation, except for the guards. Similar rules exist to transform the connections *acquires, imports* and *contains*.

6.4.3 Merge helper view

The *helper View2SUM* transformation is used to transform the helper view to the temporary SUM model. In order to transform this view, all the entities and connections of the helper view are transformed to similar entities and connections of the temporary SUM model. This transformation is almost identical to the *helper_SUM2View*, except for the source and target models. The complete *helper View2SUM* transformation can be found in Appendix G.
6.5 Merge SUM transformation

The merge SUM transformation is the last step in the transformation chain, and synchronizes the SUM with the previously edited views. The temporary SUM model, which is created from either the context\_View2SUM, struct\_View2SUM or helper\_View2SUM transformation, is used as input model. Together with the old SUM model, these models are transformed into a new SUM model. The new SUM model contains all the changes that have been made to the views during the transformation chain. This transformation was already provided by nAOMi, and no major adjustments to the code have been made. However, in order to describe the complete transformation chain, this transformation will also be discussed in detail. The complete merge\_SUM transformation can be found in Appendix E.

The transformation uses two models as input models, the temporary SUM model which contains view model entities adhering to the SUM metamodel, and the old SUM model which contains all the entities of the system, also adhering to the SUM metamodel. The result of the transformation is a new SUM model that contains all the entities of the system, including any updates or changes that are made in the view. For every entity of both the old SUM and the temporary SUM, a guard has been added to check whether the entity should be transformed. The helpers that are used by this guard are shown in Listing 6.7.

```
1 helper def : eOfIN1 : Set(String) = PLM1!Element->allInstancesFrom('IN1');
2 helper def : eOfIN2 : Set(String) = PLM1!Element->allInstancesFrom('IN2');
3 helper def : idOfIN2 : Set(String) = PLM1!Element->allInstancesFrom('IN2')->collect(e|e.xmiID->
4   =~);
5
6 helper def : toTransform(s : OclAny) : Boolean = (thisModule.eOfIN2->includes(s))
7   xor (not thisModule.idOfIN2->includes(s._xmiID_) and thisModule.eOfIN1->includes(s));
```

Listing 6.7: Merge\_SUM: toTransform helper

Helpers `eOfIN1`, `eOfIN2` and `idOfIN2` define a set of `elements`. Helper `eOfIN1` contains a set of elements from the old SUM model, `eOfIN2` contains a set of elements from the temporary SUM model, and `idOfIN2` contains a set of `xmiIDs` of elements from the temporary SUM model. Each entity, connection, attribute etc. in a model has a unique `xmiID`, which is used to check whether they are exactly the same, or just, e.g., have the same name.

Helper `toTransform` is the actual check to see whether we need to transform, and thus include, the entity in the new SUM model. The statement in lines 6 and 7 of Listing 6.7 is equal to the following logical statement

\[(X \ xor \ ((not \ Y) \ and \ Z))\]

This means that either \(X\) should be true, or \(Z\) should be true and \(Y\) should be false. Translated to the code of Listing 6.7, this means that the entity should always be part of either the old SUM or the temporary SUM. However, when the entity is part of the old SUM its `xmiID` should not be contained in the set of `xmiIDs` of the temporary SUM. Since every entity has an `xmiID`, the entity will be transformed when it is in the old SUM, in the temporary SUM or in both. The check for the `xmiID` is done to prevent merging new entities that have the same `xmiID` as identities in the old SUM, but are not the same entity.

This transformation is structured slightly different from the other transformations. In stead of using normal rules to match all the connections, unique lazy rules are used. Within the do-section of the entity rule, we set the connections of the entity, with the use of a unique lazy rule. Since every connection is connected to an entity, all connections will be transformed in this way. Using this method, there is no need for a guard on the connection as well. This way all the connections that are connected to an entity that shouldn’t
be transformed are also not transformed. The participations and attributes of the entities and connections are set using helpers and unique lazy rules, like in the previously discussed transformations. Finally, the \textit{merge\_SUM} transformation establishes the instance-type relationships between the entities and connections at different levels.

\section*{6.6 Conclusion}

In this chapter we discussed the transformation chain of the models. We first introduced ATL, and explained the main concepts of this language to better understand the example code used in this chapter. Next we introduced the transformation chain, showing all the transformations used in our current implementation of nAOMi. Each transformation has been discussed individually, using examples from the transformation code. We started with the generate view transformations, which is the first step of the transformation chain. Next we looked into the merge view transformations, that transform the views into the temporary SUM. Finally we discussed the last step of the transformation chain, the merge SUM transformation. This transformation synchronizes the old SUM model with the temporary SUM model, resulting in the new SUM model. After the final step, the complete process can be repeated. This enables an ongoing development of the different views and the SUM system.
Chapter 7

Implementation

In this chapter we discuss the implementation of the extension to nAOMi. Most of the implementation regarding the views and transformations have already been discussed in Chapter 5 and 6 respectively. However, in order to connect these models and transformations, we need to extend the tool nAOMi, which is discussed in this Chapter. First we discuss the Eclipse plug-ins and how we extended nAOMi using the provided extension points. Furthermore, we look into the dimension configuration file that plays an important role in the implementation of our extension.

7.1 Eclipse plug-ins

nAOMi has been developed as an Eclipse plug-in project, which fosters modular extensions. Using this method, the program can be easily extended with additional modules that don’t need any alterations to the original code. In this section we describe the plug-ins that are used to support our additional view.

Eclipse plug-in extensions consist of two different types, the plug-in with the extension point, and the extension plug-in [36]. The former provides a so called extension point, which can be used to extend a project, whereas the latter links to this extension point in order to build the extended project. In order to create an extension point in a project, the MANIFEST.MF file has to be edited. Within this file in the Extension Points tab, it is possible to define an extension point. In order to create the extension point, a name, ID and scheme has to be defined. The scheme is an XSD scheme in which the properties and elements of the extension point can be defined. The left side of Figure 7.1 shows an example of the elements and attributes defined in the sum extension point. Every element in the extension to this plug-in should be defined according to the specifications that are set in the extension point. For example, when a transformation is added to the extension plug-in, it has to have a ViewToSum and a SumToView element, a class, an id, and a name. Furthermore, there are other elements such as Options and InModel that contain attributes and other elements. The elements are built up as a tree structure as can be seen from the extensions on the right of Figure 7.1. In order to create an extension plug-in, one has to add an extension point in the extensions tab to the MANIFEST.MF file. In Figure 7.1 we see that the extension point de.uni_mannheim.informatik.swt.naomi.trafo has been added. For this extension we can now define the elements and attributes as created in the extension point. The extension of Figure 7.1 contains two transformations, one for the structural view and one for the context view. For
each of these transformations all the attributes and elements have to be set. Each transformation contains all
the information that is necessary to use the transformation, such as the models, metamodels, transformations
and the file locations of these.

In order to add a view and extend the SUM metamodel, there are two plug-ins that should be extended, the
sum plug-in and the trafo plug-in. The extension of the trafo plug-in has been described above, and defines
the transformations of the view. The extension of the sum plug-in is defined in the same way but instead of
the transformations it defines the properties of the SUM and the dimension configuration file.

7.1.1 Transformation plug-in

The transformation plug-in contains all the transformations as described in Chapter 6. Furthermore it con-
tains the metamodels for the context view and the structural view. Besides the models and the transfor-
mations, this plug-in also implements the java interface that handles the transformations. This interface
contains two methods, one for the generation of the views, called transform(...) and one for the merge of the
view, called merge(...). Both of these methods are implemented in the extension and make sure that the right
transformation is launched. The methods get the transformation properties, as specified by the configuration
file, provided as parameters. After which it transforms all these properties to options that can be used in the
ATL transformation launcher. For every transformation the InModels, OutModels, Paths and Options have to
be set. The class also contains a filename resolver, that looks for the right file according to the given param-
eter. The string [sum_mm] will resolve to the SUM metamodel file location that is specified as an attribute
in the class. For example, this [sum_mm] will resolve to sumBundle.getEntry("/sum/SUM_MMXlml");, where
sumBundle is nl.utwente.naomi.extension.sum. For the merge transformation we not only set all the
options, but also make sure that the SUM_Merge transformation is launched directly after the View2SUM
transformation. Since the View2SUM transformation creates a temporary SUM file, this file will be deleted
after the SUM_Merge transformation.
7.2. DIMENSION CONFIGURATION

7.1.2 SUM plug-in

The SUM-plug-in contains the SUM model, its metamodel and the dimension configuration file, which will be discussed in the next section. By defining the SUM model and metamodel as a complete new model for our extension, the original SUM model is no longer used. This meets the requirement that the original SUM model should not be modified. Furthermore, the extension of this plug-in automatically creates a new option in the new nAOMi project wizard provided by nAOMi. The code responsible for this wizard automatically checks how many extensions are present for the SUM plug-in, and creates a project option for every one of them. This allows us to use both the native nAOMi functionality as well as our extended version.

7.2 Dimension configuration

The dimension configuration file is a configuration file that configures the content and properties of the selectable dimensions. This configuration file has an XML style and contains elements in a tree shape. With this configuration file part of the handling of the selected items is configured. The main elements of this file are discussed below. The complete dimension configuration file has been added as Appendix H.

- **DimensionContainer** This container contains all the dimensions that should be present in the tool. In this case there are seven different dimensions, namely the version, abstraction, component, encapsulation, projection, granularity, and operation. Each of these dimensions specify the content that is shown within these containers. For some of the dimensions a `contentProvider` has been added, which is specified in the `contentProviderContainer`. It is also possible to create `dimensionElements`, as done in the `projection` dimension, to support the different views (lines 8 to 13, Listing 7.1). Another option is to use a combination of the content, by adding a content provider as well as a hard-coded element. This is done in the `component` dimension, in lines 1 to 3. All the components of the SUM model are added by using the `contentProvider`, whereas the complete overview option is added with a `dimensionElement`.

```xml
<dimension name="Component" icon="icons/Component.png" contentProviders="/contentProviderContainer/@contentProvider.1" contentComparator="/contentComparatorContainer/@contentComparator.1">
  <dimensionElement name="Complete Overview" defaultSelection="true"/>
</dimension>
<dimension name="Encapsulation">
  <dimensionElement name="Specification" defaultSelection="true" icon="icons/blackbox.png"/>
</dimension>
<dimension name="Projection" icon=""/>
  <dimensionElement name="Structural"/>
  <dimensionElement name="Behavioural"/>
  <dimensionElement name="Contextual"/>
  <dimensionElement name="Helper"/>
</dimension>
```

Listing 7.1: Part of the dimensionContainer in the configuration file

- **MappingContainer** The mapping container maps the different selected dimensions to an editor and a transformation. This enables the possibility to use multiple editors and transformations if desired.
For example, by selecting the same view but a different encapsulation, a different transformation can be used. Listing 7.2 shows how such a mapping looks like. In this example, the mapping is mapped to the first transformation and the first editor, as seen in line 1. In order to be mapped to this transformation and editor, the dimensions specified with the \texttt{dimElSpec} should be selected. In Listing 7.2 we see specifications for all dimensions except dimensions 0 and 2. This means that the selection of these dimensions have no impact on the mapping. For the other dimensions we see that for all except \texttt{dimension 1}, \texttt{dimensionElement 0} should be selected. For \texttt{dimension 1}, \texttt{dimensionElement 1} has to be selected. When all the right elements have been selected the specified transformation and editor will be used. The meaning of the different \texttt{dimensions} and \texttt{dimensionElements} are added as comments behind each statement.

\begin{verbatim}
<mapping transformation="/ / transformationContainer / @transformation.0" editor="/ / viewEditorContainer / @editor.0"/>
<dimElSpec dimElementName="/ / dimensionContainer / @dimension.1 / @dimensionElement.1"
    dimName="/ / dimensionContainer / @dimension.1"/>
<dimElSpec dimElementName="/ / dimensionContainer / @dimension.3 / @dimensionElement.0"
    dimName="/ / dimensionContainer / @dimension.3"/>
<dimElSpec dimElementName="/ / dimensionContainer / @dimension.4 / @dimensionElement.0"
    dimName="/ / dimensionContainer / @dimension.4"/>
<dimElSpec dimElementName="/ / dimensionContainer / @dimension.5 / @dimensionElement.0"
    dimName="/ / dimensionContainer / @dimension.5"/>
<dimElSpec dimElementName="/ / dimensionContainer / @dimension.6 / @dimensionElement.0"
    dimName="/ / dimensionContainer / @dimension.6"/>
</mapping>
\end{verbatim}

Listing 7.2: Example of a mapping in the configuration file

In our configuration file we have two of these mappings, one for the structural view and one for the context view. Both of these use the same editor, but different transformations.

- **TransformationContainer** The transformationContainer contains all the properties of the transformations. These properties look like the properties that are specified in the \texttt{MANIFEST.MF} file of the extension plug-in project, as seen in Figure 7.1. For every transformation, two different transformation files have been provided. One for the transformation from the SUM to the view, and one for the transformation from the view to the SUM. For each of these transformations the input and output models and metamodels are specified.

- **ViewEditorContainer** The viewEditorContainer specifies which editor should be used to visualize and edit the views that are displayed, and the file extension that is associated with this view. In this case we use the LML diagram editor and the .lml file extension.

- **ContentProviderContainer** This element provides the content for different dimensions. Currently there is a content provider for the version, components, and the operation. However, the component provider is currently the only functional content provider in our extension.

- **ContentComparatorContainer** This element contains content comparators that compare certain aspects of the content. In this case there is a comparator for the name and for the version of the selected content.
7.3 Conclusion

In this chapter we discussed the implementation of the extension to nAOMi. One of the goals was to extend nAOMi in a modular way, such that the original code would stay untouched. By using the extension points provided by nAOMi, this goal could be achieved rather easily by creating our own plug-in extensions. The SUM and the transformation projects had to be extended to support our additional view and extended SUM. By extending the SUM plug-in and creating a complete new SUM model file, the original SUM model did not have to be altered. This solves the problem that we can not change the original SUM model. Furthermore, the automatic detection of extensions to the SUM plug-in allows us to use both the native nAOMi, as well as our own extended version of nAOMi. Also a configuration file is defined to link the selected dimensions with the transformations that should be performed accordingly. Within this dimension configuration file it is possible to specify how the dimension explorer looks like, as well as mapping different selections to a transformation and editor.
Chapter 8

Case study

In this chapter we test our extension with the use of a case study. With this case study we will walk through the development process of a SUM-based system using the two views that have been implemented and extended. Step by step we explain the decisions and show examples of the resulting models. First of all the case study itself will be discussed, with regard to the requirements of the system and the stakeholders. After that we design the context view with the DSL. Next we create the structural view and in the end we link the views together. After the model has been completed, we check whether all the requirements are met. Finally, we discuss the advantages and disadvantages of our extension and which parts can still be improved.

8.1 Overview

The case study consists of modelling a SUM-based system that uses both the context as well as the structural view. For this case study we will model a holiday booking system. This system will be a simplified version of systems that are used to book holidays, such as Sunweb [2] or Booking.com [1]. Before the complete case study is explained, the requirements for both the case study and the system that will be developed are discussed.

8.1.1 Requirements

In order to develop a system that uses all the capabilities of the extension, the case study requires to have certain specific properties. Both the context view and the structural view will be tested completely and therefore we need to make sure that all the possible entities and relations are present in the system of the case study. Furthermore, it should represent a real case scenario that is not too complex, since the example models should be easily understood. Finally the system should contain a DSL representation that visualizes the context view in the standard representation. All these requirements are listed below.

1. The system has to contain at least two subsystems.
2. The system has to contain one or more components per subsystem.

3. The system has to contain actors that interact with two or more subsystems.

4. The system has to contain actors that interact with each other.

5. The system has to contain actors that interact with only one subsystem.

6. The system has to contain all the available connections between components.

7. The system has to represent a real case scenario which is not too complex.

Besides the requirements for the case study, the actual system that is being developed also has several requirements. The important requirement that the system should not be too complex, directly results in the number of requirements for the actual system. The requirements listed below are specified to keep the system small but with just enough functionality to make it a real case scenario that adheres to the case study requirements listed above.

A. The system should be able to generate reports, based on the booking or payment information.

B. The system should have a comment function for rating and support.

C. The system should support a function that allows data to be added, deleted or updated.

D. The system should have a search function.

E. The user should be able to book a holiday.

F. The user should be able to view the current status of his booking.

G. The user should be able to pay for his holiday using the system.

### 8.1.2 Description

All the requirements from both the case study and the system, will be modelled. The system that will be developed is a holiday booking system that allows users to book their holidays. During this booking there are several steps that are performed. A user uses the system to search for a holiday, after which it can look into the specifics and comments on this holiday. Furthermore it is possible for the user to generate a summary report about this holiday, which can be sent to his email. When the user decides he wants to book the holiday, the user can do so by using the system. The system hereby handles the payment of the holiday and sends a confirmation report with information when everything has been paid. In case the user has some problems or questions he can use the comment system to contact support. Finally there is a user that is allowed to add, update or delete information from the system.
8.1. OVERVIEW

8.1.3 Actors

From requirements 3, 4 and 5 we see that we need at least two actors: one that interacts with multiple subsystems, one that interacts with only one subsystem, and both should then interact with each other. Looking at the description of the holiday booking system we can see that there are several actors that interact with the system. The easiest to distinguish is the user who wants to book a holiday. We refer to this user as the customer. Furthermore there needs to be someone that can support the customer when there are problems. This actor we call the support employee. Finally, we need a user that can add, update and delete information from the system. This stakeholder we call the administrator, who will also perform the maintenance on the system.

Subsystems

Besides the actors we need to define at least two subsystems. From the requirements we can see that there is some distinct functionality within the system. In order to keep the system simple we decided to create three different subsystems, namely file report system, booking system and financial system. Furthermore, there is a database that will be modelled as an external system. This leaves us with the following subsystems that will be shortly described below.

- **File report system**: This subsystem is based on requirement A, and allows the generation of reports. This can be a booking overview or a financial overview.

- **Financial system**: This subsystem is based on requirement G, and contains all the functionality that handles the financial part of the booking system. This includes the payment and the connection with the bank.

- **Booking system**: This subsystem is the core of the system. It is based on requirements B-F and requirement 7. In order to keep the system simple this subsystem handles most of the booking functionality. In order to meet requirement C, it communicates with the external database system.

Since the database system is modelled as an external system that interacts with the holiday booking system, it will be modelled as one of the actors. Furthermore, the financial subsystem is an external system that communicates with the bank, which will also be modelled as an actor. Considering also the bank and the database system this results in following actors, which will be described shortly.

- **Customer**: This actor interacts most with the system. The customer can book a holiday, look at the status, and comment. Furthermore, the customer interacts with the bank to pay for the holiday.

- **Support Employee**: This actor interacts with the system by using the comment system in the booking subsystem. Furthermore, the support employee can check different parts of the system such as the status of the customers booking.

- **Administrator** The administrator is allowed to add, change and update the information in the system. Therefore, the administrator interacts with the booking subsystem where the information maintained. Indirectly, the administrator will change the data in the external database system, but the administrator
never communicates with the database directly.

- **Database system** The database system contains all the data of the holiday booking system, but is not modelled as part of the system. Therefore, it is maintained by external actors and only communicates with the holiday booking system.

- **Bank** This actor handles the final payment of the booked holiday. The bank interacts with the financial subsystem as well as with the customer.

With these actors and subsystems we meet all the case study requirements that are defined in Section 8.1.1.

### 8.2 Constructing the context model

Since all the requirements have been defined, and the stakeholders and subsystems have been identified, the development of the system can be started. We start by modelling the context view since this view is often used during the requirements specification. Section 8.1 described the initial requirements specification, but by modelling the system with a context diagram, one can easily detect missing information which is one of the powerful aspects of context diagrams.

Before we start we need to make sure we start with a new SUM model. In order to do this we simply need to create a new project using our extended SUM model. Due to some problems with loading a complete empty view, there are already two entities present in a new project. There is one component and one system, both called main. These entities can be used as the base of the model, or can be edited or deleted later. We start by selecting the complete overview and the context projection. After generating this view we will see one system entity as shown in Figure 8.1.

![Figure 8.1: New SUM project with one system entity.](image)

In order to add new entities or connections to the model, the toolbar at the right side can be used to create instances from the O0 level, on which the metamodel is shown. In case of the context view we can add systems, subSystems, actors and instantiated connections. We will first add the three subsystems, followed by the five actors as discussed in Section 8.1.3. When an element has been added, it is directly shown in the DSL representation. Sometimes it can be convenient to switch to the LML representation during the development. This can be achieved by clicking on the D \ G button on the top right, when hovering the element.
After all the entities are added, we can start with the connections between those entities. First of all we need to connect subsystems with the system, by instantiating the ConsistsOf connection. When the Instantiate connection element is selected, it can be used to draw a connection from one entity to another. Upon releasing the mouse button a small window pops up that allows us to select an instantiated connection that is available for those two elements.

The result of the context view can be seen in Figure 8.2. In this figure we see all the (sub)systems, actors and relationships between them. The different relations between the actors and the subsystems are based on the requirements and the descriptions in Section 8.1.1. There could be many more of these relationships, but in order to keep it clear we chose only to include those of Figure 8.2. All the relationship connection between the entities have been collapsed, which means they are represented by small square dots.

![Diagram](image.png)

**Figure 8.2: Context view of the holiday booking system.**

After this model has been created it is time to save it and merge it with the SUM model. This is done by clicking on the merge button. It will now perform the transformations discussed in Chapter 6, and create a new SUM model with all the entities and connections that we just added.

### 8.3 Constructing the structural model

Now that the first version of the context view has been added, we can start on the structural view. The structural view contains all the components of the system and the different associations between them. A structural view is useful to show the structure of the complete system. Looking at requirement 2, from Section 8.1.1. we see that every sub-system should have at least one component. This means that, with the three sub-systems, we need at least 3 components. However, to make the case a bit more realistic a few more components will be added. In order to start constructing the structural view we need to select the complete overview and the structural view, for the component and projection respectively. When we generate this view we get the model as shown in Figure 8.3.
We can now start constructing the structural view by adding more components to the view and connecting them together with either of the four connections, creates, acquires, imports or contains. In order to keep the model simple, only a few components per sub-system will be introduced. Each component is briefly described below.

- **Booking**: This component is part of the Booking System, and is the central point of the structure. It contains the functionality to book the holiday, and to check the booking status.

- **Comment**: This component contains the functionality for the comment system, and is part of the Booking System.

- **Search**: This component is part of the Booking System, and contains the search functionality of the system.

- **Data module**: This component contains all the functionality to show and update the data of the holiday booking system. It is part of the Booking System sub-system.

- **Report**: This component is part of the File Report System, and contains the functionality to generate the report. The report is created upon request by the booking component.

- **Send report**: This component contains the functionality to send the report to the user, and is created by the report component. It is part of the File Report System.

- **Payment**: This component is part of the Financial System, and handles all the payment business of the holiday booking system.

The result of adding these components to the structural view can be seen in Figure 8.4. The main component has been replaced by the booking component which now plays the central role in the structure. The structural view is always shown in the LML representation and has no specific DSL connected to it. The connections between the components are currently not collapsed and shown as flattened hexagons. This representation is the same as the connections of the context view when not collapsed.

Now that the structural model has been completed we can save and merge it with the SUM model. By doing so, the merge transformations will be performed, as described in Chapter 6.
8.4 Linking the views

At this point two models have been created, the context model and the structural model, however they are not linked together yet, which means we can not use the SUM optimally. In order to link the views together, we use the special helper view. This view is designed to link the sub-systems of the context view with the components of the structural view. With the *includes* connection we can connect the different entities together. Figure 8.5 shows the result, with the sub-systems of the holiday booking system connected to the components.

After all the sub-systems and components are connected, the view can be merged with the SUM-model. The SUM model now contains all the information about the system, which includes the context model, the structural model and the connection between these models. The first version of the holiday booking system has been created. Whenever we would like to edit a part of the model we could simple regenerate any of the views and edit or add a part to the model. After the the edition is completed the user can merge the view and everything will be saved in the SUM model.

Since some models can grow rather big, it is convenient to have the system modelled with several sub-systems. This way only a part of the model will be generated which gives a clearer overview. Besides, you can select a single component that only shows that component and the components directly related to this component. All the different possibilities are listed below.
8.5 Conclusion

In this chapter we tested the utility and functionality of the extension, by designing a case study which contained the development of a simple system. First the requirements of the case study were discussed, after which the case study could be defined. For this case study we had to develop a *holiday booking system* that enables customers to book holidays. After the requirements for this system were defined, the sub-systems
and actors could be identified. The next step was to develop the context model, where all the important aspects are discussed. The result was a complete context model that represented the holiday booking system. After the context model was finished, we could start developing the structural model. Finally we had to link the two models together, which is done by using the helper view. The helper view allows us to connect the components of the structural view with the sub-systems of the context view. This case study provided a good testing scenario for the implemented extension because it made use of all the added functionality. This gave us a good impression on how the tool works and shows the utility of this extension.

8.5.1 Current limitations

Although most of the functionality worked, there are still some limitations in the current extension of nAOMi. The actors are, for example, all modelled with a square box and specify their type with an attribute, whereas it would be better to create a visual distinction between them as well. The humanoid entity could then be represented by a stick figure, whereas the systems can be represented as a square box. Furthermore, the lines are straight and angular whereas the intended context representation uses curved lines. Another functional part is that the connection between the components and the sub-systems are now done with the helper view. This works, but it would be better if we could specify the sub-system of the component as an attribute. This would eliminate the helper view and create a cleaner solution. Finally the context view is now always shown in the DSL representation. It would be better to have the ability to choose between these options, using the abstraction dimension. All together, the solutions to these limitations will create a better and cleaner extension, but are not required to use the functionality that has been added.

Finally there is one limitation about the generated views in general. After a view has been generated all the components and connections appear at the left top side of the model. This means they all overlap each other, and the user has to restructure the model manually. This is rather time consuming and a solution for this problem should therefore be found in the future.

8.5.2 Other cases

Although this case study examines a specific case, we believe that the extension of the context view also helps other kind of systems. All systems that are developed interact in some way with other entities. Most of the systems interact with both stakeholders and other systems, but some may only interact with systems. No matter which kind of actors they interact with, it is always useful to construct a context view, as it defines the boundaries of the system. Even if there are only one or two actors, because the developers have to think about the system boundaries, which gives them more understanding of the system they are developing.
Chapter 9

Conclusion

In this thesis we investigated the topics of multi-level modelling and SUM-based modelling and the practical use of these techniques. The main goal of this thesis is to show the applicability of these techniques by extending the multi-level SUM-based modelling tool nAOMi with an extra view. Before this extension could be developed, both modelling techniques and the available tools had to be studied thoroughly. We defined the design requirements (Section 1.2.1.) and had to make sure the techniques covered enough functionality to meet these requirements.

We started with a literature study of multi-level modelling and SUM-based modelling. Chapter 2 showed that the current MDA still has several limitations that are solved by the new multi-level modelling technique. Multi-level modelling is a technique based on the principles of MDA, but introduces the concept of multiple levels and distinguishing the ontological and the linguistic layers. The PLM and the LML were introduced and discussed to give a clear overview on how these models could be used in practice. In chapter 3 another modelling technique was introduced, SUM-based modelling. This modelling technique is based on view-based modelling, but uses a single underlying model that contains all the information about the system, which can be used to generate views. After discussing the functionality of this technique, we looked into the proposed idea by Atkinson et al. to combine the two techniques of multi-level modelling and SUM-based modelling. During this research we found two tools that were useful for combining these two techniques, namely Melanee and nAOMi. Melanee is a tool that supports the design and transformation of multi-level models. nAOMi is a SUM-based tool that uses Melanee to represent the models. Especially the latter tool was interesting for the practical use of combining these techniques. Both of these tools have been thoroughly examined and we concluded that, although nAOMi is still in an early stage of development, it was suitable for extending the tool with an extra view.

The literature study of the modelling techniques and the tools allowed us to answer the first research question, "What is the applicability of multi-level SUM-based modelling?" (RQ1). This research question consists of three sub-questions, namely "What is the coverage of multi-level modelling w.r.t. our design requirements" (SQ1.1), "What is the coverage of SUM-based modelling w.r.t. our design requirements?" (SQ1.2), and "How stable and complete are the available tools?" (SQ1.3). After studying multi-level modelling and SUM-based modelling, we concluded that the coverage of those techniques is sufficient enough to meet our design requirements. The techniques contain the functionality to design and transform multi-level models, which are the most important aspects needed for our extension. This answers our first two sub-questions
SQ1.1 and SQ1.2. In order to answer sub-question SQ1.3 we had to examine the tools Melanee and nAOMi. During this process we found that Melanee is stable and contains a lot of functionality that allows us to create multi-level models. However, nAOMi is still in an early stage of development and is less stable and complete. Although there are some limitations in nAOMi, we concluded that the tool was complete enough to develop an extension. After answering the three sub-questions we concluded that the multi-level SUM-based modelling provides a good foundation and is applicable in practice.

The second research question read "How can we extend nAOMi with a view in a modular way?" (RQ2). This research question consists of two sub-questions, namely "How do we extend the SUM metamodel without affecting the original metamodel?" (SQ2.1), and "How do we represent the view in a DSL visualization?" (SQ2.2). In order to answer these questions we had to extend the tool nAOMi with a view that meets the design requirements. In Chapter 5 we started with searching for a useful view that met these requirements, and discussed two of them in detail. Ultimately the context view was chosen. This view contributes to the system development by specifying the boundaries of the system and visualizing the interactions with its stakeholders. In order to answer SQ2.1 we had to find a way to represent this view in its intended visualization. This could be achieved by using the visualization editor, provided by Melanee. With this tool we could specify a custom representation for every entity or connection of the metamodel. Whenever the view would be generated it would be represented using the custom DSL. In Chapter 6 we discussed the transformations that are used to transform the views from and to the SUM model. Each model contains two different transformations, one for the generation of the view and one to merge the view into a temporary SUM. Furthermore, there is one transformation that merges the temporary SUM with the old SUM, in order to create the new SUM model. In chapter 7 we discussed the implementation of the extension in nAOMi. Because nAOMi is developed as an Eclipse plug-in, we could easily extend the tool in a modular way, by using the provided extension points. For our extension we needed to extend two projects, one that contained the SUM models and the dimension configuration, and one that contained all the transformations and view metamodels. This answered SQ2.1, since it solved the problem of extending the SUM model without editing the original model. Furthermore, it answered RQ2 because the complete extension could be modelled in a modular way.

Finally we presented a case study to test and show the utilization of the extension. For this case study a holiday booking system was designed from scratch using nAOMi and our extensional view. This case study was chosen because we believe it represents a real case scenario, and the process would be similar for most systems. By performing this case study step by step, we discussed and tested all the functionality of our extension.

9.1 Limitations

Since nAOMi and the technology for combining multi-level modelling and SUM-based modelling are still in an early stage of development there are still some limitations. At the moment this thesis started, nAOMi only supported a small part of the structural view, which had to be extended in order to make it useful. Furthermore, it lacked some functionality regarding a dynamic custom DSL. It is, for example, not possible to determine the visual representation based on the attributes of the entities. Besides, the lines used to connect entities are always shown in a straight angular way, while sometimes a curved line is preferred. Finally, a big limitation in nAOMi is messy the view after generating a model. After generation all the entities and connections are overlapped in the top left corner, and the user has to manually drag them into
the right positions. Besides these limitations, nAOMi provided a good foundation for the extension of the SUM model and views. Also the use of the dimension configuration file gave many options to alter the selection menu of the tool.

With regard to our own extension there are also some limitations that should be mentioned. The foremost would be the connection between the two views. Currently the helper view needs to be used to connect the components and the sub-systems. Furthermore, the context view is always shown in the DSL representation. Although, this is probably the preferable option, it can sometimes be convenient to use the default LML visualization.

All together we created a solid extension that meets all the design requirements and answers the research questions as stated in Chapter 1. Although some of the functionality of the extension could still be improved, the extension enables the utilization of the context and structural view. With this extension we also showed that the multi-level modelling and SUM-based modelling techniques and the tools nAOMi and Melanee are complete enough to be used in practice. This brings us one step closer to use multi-level SUM-based modelling for modelling complete systems.

9.2 Further research

In this thesis we researched the current applicability of multi-level SUM-based modelling, and developed an extension for the nAOMi tool. By implementing this extension we showed that there is currently a good foundation to support multi-level SUM-based modelling. nAOMi is currently the only tool that supports both of these techniques, and by extending this tool we contributed by showing the utilization of this tool. By looking at the limitations, the tool itself still requires a lot of work to create a fully functional and practical tool. At the moment it is a good tool for conducting research, but not complete enough to really model complete systems. With our extension there are two supported views, while many more could be developed. During our view selection we also looked into the possibility to add a sequence view. If the DSL editor of Melanee could be extended with some extra functionality, the sequence view would be a useful addition to nAOMi. Besides, extending the DSL editor would also favour the visualization of other views.

Furthermore, not all dimensions of the dimension explorer are currently used. It would therefore be a good idea to extend the options of the views such that all the dimensions can be used for every view. This would also adhere more to the Kobra design methodology.

Another idea could be to support not only the multi-level models but also the standard UML models. This way the developers that are familiar with UML modelling are more likely to switch to multi-level modelling. By enabling the development of UML models and the transformation to the SUM, the UML models could then automatically be generated in the LML representation.
References


owl2-overview/.


Appendices
Appendix A

Struct_SUM2View

This transformation contains the code to generate the structural view from the SUM model.

```
module Extended_SUM2Struct;
create OUT : STRUCT from IN : SUM, parameters : Parameters;
helper def : getParameter(name : String) : String =
  Parameters!Param->allInstancesFrom('parameters')-->
  select (e | e.name = name)-->
  first ().value.

--pre-defined '-Complete Overview-' shows all the components and connections of the system.
helper def : all : String = '-Complete Overview-';
helper def : subjectName : String =
  thisModule.getParameter('Component').debug('selectedComponent');

-- Get all acquires/creates/imports/contains relations that are connected to the subjectComponent
helper def : acq : OclAny = SUM!"O0.Acquires"-->
  allInstancesFrom ('IN')-->
  select (i | i . _ . name =
    thisModule.subjectName . debug('subname')-->
    first ().debug('first'));
helper def : cre : OclAny = SUM!"O0.Creates"-->
  allInstancesFrom ('IN')-->
  select (i | thisModule.co
    on(i));
helper def : imp : OclAny = SUM!"O0.Imports"-->
  allInstancesFrom ('IN')-->
  select (i | thisModule.con
    on(i));
helper def : cont : OclAny = SUM!"O0.Contains"-->
  allInstancesFrom ('IN')-->
  select (i | thisModule.
    con(i));

-- Get all the components that are connected to the subjectComponent by an acquire connection.
```

83
rule Component {
  from
  s : SUM! "O0. Component" ((thisModule.com -> includes(s) or (s = thisModule.sub))) or (thisModule.
  subjectName = thisModule.all))
to
  t : STRUCT! "O0. Component" ( 
    ".xmlID" <- s."xmlID",
    "name" <- s.name,
    "feature" <- thisModule.createMethods(s)
  )
}

rule Acquires {
  from
  s : SUM! "O0. Acquires" ((thisModule.acq -> includes(s)) or (thisModule.subjectName = thisModule.
  all))
to
  t : STRUCT! "O0. Acquires" ( 
    ".xmlID" <- s."xmlID",
    "name" <- s.name,
    "feature" <- Sequence{thisModule.createMethods(s),thisModule.createAttributes(s)},
    "participation" <- (s.getAllParticipations () -> collect (e | thisModule.
    createParticipations(e)))
  )
}

rule Creates {
  from
  s : SUM! "O0. Creates" ((thisModule.cre -> includes(s)) or (thisModule.subjectName = thisModule.
  all))
to
  t : STRUCT! "O0. Creates" ( 
    ".xmlID" <- s."xmlID",
    "name" <- s.name,
    "feature" <- Sequence{thisModule.createMethods(s),thisModule.createAttributes(s)},
    "participation" <- (s.getAllParticipations () -> collect (e | thisModule.
    createParticipations(e)))
  )
}

rule Imports {
  from
  s : SUM! "O0. Imports" ((thisModule.imp -> includes(s)) or (thisModule.subjectName = thisModule.
  all))
to
  t : STRUCT! "O0. Imports" ( 
    ".xmlID" <- s."xmlID",
    "name" <- s.name,
    "feature" <- Sequence{thisModule.createMethods(s),thisModule.createAttributes(s)},
    "participation" <- (s.getAllParticipations () -> collect (e | thisModule.
    createParticipations(e)))
  )
}

rule Contains {
  from
  s : SUM! "O0. Contains" ((thisModule.cont -> includes(s)) or (thisModule.subjectName = thisModule.
  all))
to
  t : STRUCT! "O0. Contains" ( 
    ".xmlID" <- s."xmlID",
    "name" <- s.name,
    "feature" <- Sequence{thisModule.createMethods(s),thisModule.createAttributes(s)},
    "participation" <- (s.getAllParticipations () -> collect (e | thisModule.
    createParticipations(e)))
  )
lazy rule createParticipations {
  from
  s : SUM! Participation
  to
  t : STRUCT! Participation (  
    _xmlID_ <= s._xmlID_,
    destination <= s.destination,
    lower <= s.lower,
    upper <= s.upper,
    navigable <= s.navigable,
    containment <= s.containment,
    connection <= s.connection,
    roleName <= s.roleName
  )
}

helper def : createMethods(s : SUM! CObject) : Sequence(STRUCT! Method) =
  s._1_.feature -> iterate(i; ret : Set(STRUCT! Method) = Sequence{}) |
  if (i -> oclType() -> conformsTo(SUM! Method))
    then ret -> including(thisModule.createMethod(i))
  else ret
  endif;

helper def : createAttributes(s : SUM! CObject) : Sequence(STRUCT! Attribute) =
  s._1_.feature -> iterate(i; ret : Set(STRUCT! Attribute) = Sequence{}) |
  if (i -> oclType() -> conformsTo(SUM! Attribute))
    then ret -> including(thisModule.createAttribute(i))
  else ret
  endif;

unique lazy rule createMethod {
  from
  s : SUM! Method
  to
  t : STRUCT! Method (  
    _xmlID_ <= s._xmlID_,
    name <= s.name,
    body <= s.body
  )
}

unique lazy rule createAttribute {
  from
  s : SUM! Attribute
  to
  t : STRUCT! Attribute (  
    _xmlID_ <= s._xmlID_,
    name <= s.name,
    value <= s.value,
    mutability <= s.mutability,
    durability <= s.durability
  )
}
Appendix B

Context_SUM2View

This transformation contains the code to generate the context view from the SUM model.

```plaintext
module Extended_SUM2Context;
cREATE OUT : CONTEXT from IN : SUM, parameters : Parameters;

helper def : getParameter(name : String) : String =
  Parameters!Param->allInstancesFrom('parameters')->select(c | c.name = name)->first().value;

-- pre-defined '-Complete Overview-' shows all the components and connections of the system.
helper def : all : String = '-Complete Overview-';

helper def : system : CONTEXT!"O0.System" = OclUndefined;

helper def : abstraction : String = thisModule.getParameter('Abstraction');
helper def : psm : String = 'PSM';
helper def : encapsulation : String = thisModule.getParameter('Encapsulation');
helper def : black : String = 'Specification';

-- get the selected component
helper def : subjectName : String = thisModule.getParameter('Component');
helper def : sub : OclAny = SUM!"O0.Component"->allInstancesFrom('IN')->select(i | i._1_.name = <-thisModule.subjectName>)->first();

-- Get all(normally 1) includes connections that are connected to the subComponent
helper def : inc : OclAny = SUM!"O0.Includes"->allInstancesFrom('IN')->select(i | thisModule. <-con(i));
helper def : con (c : SUM!Connection) : Boolean = ( c.getAllParticipants() -> exists ( j | <-thisModule.par(j)) );
helper def : par (p : SUM!Participation) : Boolean = (p.destination = thisModule.sub);

-- get the SubSystem that is connected to the selected component
helper def : sys : OclAny = SUM!"O0.SubSystem"->allInstancesFrom('IN')->select ( i | (thisModule. <-inc -> exists ( j | (j.getParticipants() -> exists( k | k =i )))) );

-- Get all the relationship connections that are connected to the subsystem
helper def : relf : OclAny = SUM!"O0.RelationshipFrom"->allInstancesFrom('IN')->select( i | <-thisModule.conn(i));
helper def : relt : OclAny = SUM!"O0.RelationshipTo"-> allInstancesFrom('IN')->select( i | <-thisModule.conn(i) );
helper def : reli : OclAny = SUM!"O0.RelationshipInternal"->allInstancesFrom('IN')->select( i | <-thisModule.conn(i) );
```

87
helper def : conn (c : SLIM! Connection) : Boolean = (c.getAllParticipations() -> exists (j | thisModule.part(j)));

helper def : part (p : SLIM! Participation) : Boolean = (thisModule.sys -> includes(p.destination)) -> exists (c | thisModule.conn(c));

-- get all the actors connected to the subsystem.
helper def : act : OclAny = SLIM!"O0. Actor"->allInstancesFrom(\"IN\")->select (i | ((thisModule.relf -> exists (j | (j.getParticipations() -> exists (k | k = i))))) or (thisModule.relt -> exists (j | (j.getParticipations() -> exists (k | k = i))))));

-- get connected subsystems
helper def : css : OclAny = SLIM!"O0. SubSystem"->allInstancesFrom(\"IN\")->select (i | (thisModule.reli -> exists (j | j.getParticipations() -> exists (k | k = i))));

-- Get all the relationship connections that are indirectly connected to the subsystem.
helper def : relfi : OclAny = SLIM!"O0. RelationshipFrom"->allInstancesFrom(\"IN\")->select (i | (thisModule.conn(i)));

-- Get all the relationship connections that are indirectly connected to the subsystem.
helper def : relti : OclAny = SLIM!"O0. RelationshipTo"->allInstancesFrom(\"IN\")->select (i | (thisModule.conn(i)));

-- get all the relationship connections that are indirectly connected to the subsystem.
helper def : reli : OclAny = SLIM!"O0. RelationshipInternal"->allInstancesFrom(\"IN\")->select (i | (thisModule.conn3(i)));

-- get all relations between actors that are both connected to the subsystem
helper def : rel : OclAny = SLIM!"O0. RelationshipExternal" -> allInstancesFrom(\"IN\") -> select (i | thisModule.conn2(i));

-- get relations where both actor ends are connected to the subsystem
helper def : conn2 (c : SLIM! Connection) : Boolean = (c.getAllParticipations() -> forall (j | thisModule.part2(j)));

helper def : part2 (p : SLIM! Participation) : Boolean = (thisModule.act -> includes(p.destination)) -> exists (c | thisModule.conn2(c));

-- get relations where both subsystem ends are connected to the subsystem
helper def : conn3 (c : SLIM! Connection) : Boolean = (c.getAllParticipations() -> forall (j | thisModule.part3(j)));

helper def : part3 (p : SLIM! Participation) : Boolean = (thisModule.css -> includes(p.destination)) -> exists (c | thisModule.conn3(c));

-- get relations that have an actor end and a subsystem end that is connected to the subsystem.
helper def : conn4 (c : SLIM! Connection) : Boolean = (c.getAllParticipations() -> exists (j | thisModule.part3(j)) and (c.getAllParticipations() -> exists (j | thisModule.part2(j))));

rule SubSystem { from
  s : SLIM!"O0. SubSystem" (thisModule.sys -> includes(s) or (thisModule.css -> includes(s)) or (((thisModule.subjectName = thisModule.all) and not(thisModule.encapsulation = thisModule.black))
  )
  to
  t : Context!"O0. SubSystem"("._xmIID_" <- s._xmIID_,
  "._name" <- s._name,
  name <- s.name,
  "._potency" <- s._potency,
  "._feature" <- thisModule.createMethods(s)
  )
}

rule System { from
  s : SLIM!"O0. System"(thisModule.subjectName = thisModule.all)
  to
  t : Context!"O0. System"("._xmIID_" <- s._xmIID_,

name ← s.name,
"_l_.name" ← s._l_.name,
"_l_.potency" ← s._l_.potency,
"_l_.feature" ← thisModule.createMethods(s)
}
do
{thisModule.system ← t;
}

rule Actor {
from
s : SUM!"O0. Actor" ((thisModule.act -> includes(s)) or (thisModule.subjectName = thisModule.all))
t : Context!"O0. Actor" (
"_l_.xmiID" ← s._l_.xmiID,
name ← s.name,
type ← s.type,
"_l_.name" ← s._l_.name,
"_l_.potency" ← s._l_.potency,
"_l_.feature" ← thisModule.createMethods(s)
)
}

rule RelationshipFrom {
from
s : SUM!"O0. RelationshipFrom" ((thisModule.rel -> includes(s)) or (thisModule.relfi -> includes(s)) or (thisModule.subjectName = thisModule.all))
t : Context!"O0. RelationshipFrom" (
"_l_.xmiID" ← s._l_.xmiID,
"_l_.name" ← s._l_.name,
description ← s.description,
"_l_.feature" ← thisModule.createMethods(s)
)
do
if (thisModule.encapsulation = thisModule.black)
{
t."_l_.participation" ← (s.getAllParticipations() -> collect (e | thisModule.redirectParticipations(e)));
}
else
{
t."_l_.participation" ← (s.getAllParticipations() -> collect (e | thisModule.createParticipations(e)));
}
}

rule RelationshipTo {
from
s : SUM!"O0. RelationshipTo" ((thisModule.rel -> includes(s)) or (thisModule.relTi -> includes(s)) or (thisModule.subjectName = thisModule.all))
t : Context!"O0. RelationshipTo" (
"_l_.xmiID" ← s._l_.xmiID,
"_l_.name" ← s._l_.name,
description ← s.description,
"_l_.feature" ← thisModule.createMethods(s)
)
do
if (thisModule.encapsulation = thisModule.black)
{  
  t."_\_ . participation" <- (s.getAllParticipations() -> collect (e | thisModule.  
  redirectParticipations(e)))
}
else  
{  
  t."_\_ . participation" <- (s.getAllParticipations() -> collect (e | thisModule.  
  createParticipations(e)))
}
}

rule RelationshipInternal {  
from  
  s : SUM!"O0. RelationshipInternal" (((thisModule.reli -> includes(s)) or (thisModule.relii ->  
  includes(s)) or ((thisModule.subjectName = thisModule.all)  
  and not(thisModule.encapsulation = thisModule.black))))
  to  
  t : Context!"O0. RelationshipInternal" (  
    ".\_\_ .xmiID\_\_" <- s..\_\_ .xmiID\_\_,  
    "._\_ .name" <- s..\_\_.name,  
    description <- s.description,  
    "._\_ .participation" <- (s.getAllParticipations() -> collect (e | thisModule.  
      createParticipations(e)) ),  
    "._\_ .feature" <- thisModule.createMethods(s)
  )
}

rule RelationshipExternal {  
from  
  s : SUM!"O0. RelationshipExternal" (((thisModule.rele -> includes(s)) or (thisModule.  
  subjectName = thisModule.all)))
  to  
  t : Context!"O0. RelationshipExternal" (  
    ".\_\_ .xmiID\_\_" <- s..\_\_ .xmiID\_\_,  
    "._\_ .name" <- s..\_\_.name,  
    description <- s.description,  
    "._\_ .participation" <- (s.getAllParticipations() -> collect (e | thisModule.  
      createParticipations(e)) ),  
    "._\_ .feature" <- thisModule.createMethods(s)
  )
}

rule ConsistsOf {  
from  
  s : SUM!"O0. ConsistsOf" ((thisModule.subjectName = thisModule.all and thisModule.encapsulation  
    <> thisModule.black))
  to  
  t : Context!"O0. ConsistsOf" (  
    ".\_\_ .xmiID\_\_" <- s..\_\_ .xmiID\_\_,  
    "._\_ .name" <- s..\_\_.name,  
    "._\_ .participation" <- (s.getAllParticipations() -> collect (e | thisModule.  
      createParticipations(e)) ),  
    "._\_ .feature" <- thisModule.createMethods(s)
  )
}

lazy rule createParticipations {  
from  
  s : SUM! Participation
  to  
  t : Context! Participation (  
    .xmiID.. <- s..xmiID..,  
    destination <- s.destination,  
    lower <- s.lower,  
    upper <- s.upper,  
    navigable <- s.navigable,  
    containment <- s.containment,
connection ← s.connection,
roleName ← s.roleName
)

lazy rule redirectParticipations {
  from
  s : SUM! Participation
  to
  t : Context! Participation ( 
    _xmlID_ ← s._xmlID_,
    destination ← s.destination, 
    lower ← s.lower, 
    upper ← s.upper, 
    navigable ← s.navigable, 
    containment ← s.containment, 
    connection ← s.connection, 
    roleName ← s.roleName
  )
  do
    { --redirect the participations to the system instead of the subsystem 
      if (thisModule.allSubSys -> includes(s.destination))
        { 
          t.destination ← thisModule.system;
        }
      else 
        { 
          t.destination ← s.destination;
        }
    }
}

helper def: allSubSys : OclAny = SUM!"O0.SubSystem"->allInstancesFrom('IN');

helper def: createMethods(s : SUM! ClObject) : Sequence(Context!Method) = 
  s._L_.feature -> iterate(i; ret : Set(Context!Method) = Sequence{} | 
    if (i ->oclType() -> conformsTo(SUM! Method))
      then ret -> including(thisModule.createMethod(i))
    else ret
  endif
);

helper def: createAttributes(s : SUM! ClObject) : Sequence(Context!Attribute) = 
  s._L_.feature -> iterate(i; ret : Set(Context!Attribute) = Sequence{} | 
    if (i ->oclType() -> conformsTo(SUM! Attribute))
      then ret -> including(thisModule.createAttribute(i))
    else ret
  endif
);

unique lazy rule createMethod {
  from
  s : SUM! Method
  to
  t : Context!Method ( 
    _xmlID_ ← s._xmlID_,
    name ← s.name, 
    body ← s.body
  )
}

unique lazy rule createAttribute {
  from
  s : SUM! Attribute
  to
  t : Context!Attribute ( 
    _xmlID_ ← s._xmlID_,
    name ← s.name,

  )
}
value <- s.value,
mutability <- s.mutability,
durability <- s.durability
)
}

Listing B.1: Context_SUM2View transformation
Appendix C

Struct_View2SUM

This transformation contains the code to merge the structural view with the SUM model. This results into the temporary SUM file, which will be merged with the old SUM model using the Merge_SUM transformation afterwards.

```plaintext
module Extended_Struct2SUM;

create OUT : PLM2 from IN : PLM;

rule Component {
  from
  s : PLM!"00. Component"
  to
  t : PLM2!"00. Component" (  
    _l_.xmlID_ = s._l_.xmlID_,  
    name = s._l_.name,  
    _l_.feature = thisModule.createMethods(s)
  )
}

rule Acquires {
  from
  s : PLM!"00. Acquires"
  to
  t : PLM2!"00. Acquires" (  
    _l_.xmlID_ = s._l_.xmlID_,  
    _l_.name = s._l_.name,  
    _l_.feature = Sequence{thisModule.createMethods(s),thisModule.createAttributes(s)},  
    _l_.participation = (s.getAllParticipations() -> collect (e | thisModule.creatParticipations(e)))
  )
}

rule Creates {
  from
  s : PLM!"00. Creates"
  to
  t : PLM2!"00. Creates" (  
    _l_.xmlID_ = s._l_.xmlID_,  
    _l_.name = s._l_.name,  
    _l_.feature = Sequence{thisModule.createMethods(s),thisModule.createAttributes(s)}
  )
}
```
"_l_. participation" <= (s.getAllParticipations() -> collect (e | thisModule.
    => createParticipations(e)))
}
}

rule Imports {
  from
  s : PLM!"O0. Imports"
  to
  t : PLM2!"O0. Imports" (s
    "_l_.xmlID" <= s._l_.xmlID,
    "_l_.name" <= s._l_.name,
    "_l_.feature" <= Sequence{thisModule.createMethods(s),
                        thisModule.createAttributes(s)},
    "_l_. participation" <= (s.getAllParticipations() -> collect (e | thisModule.
    => createParticipations(e)))
}

rule Contains {
  from
  s : PLM!"O0. Contains"
  to
  t : PLM2!"O0. Contains" (s
    "_l_.xmlID" <= s._l_.xmlID,
    "_l_.name" <= s._l_.name,
    "_l_.feature" <= Sequence{thisModule.createMethods(s),
                        thisModule.createAttributes(s)},
    "_l_. participation" <= (s.getAllParticipations() -> collect (e | thisModule.
    => createParticipations(e)))
}

lazy rule createParticipations {
  from
  s : PLM! Participation
  to
  t : PLM2! Participation (s
    .xmlID_ <= s._l_.xmlID_,
    destination <= s.destination,
    lower <= s.lower,
    upper <= s.upper,
    navigable <= s.navigable,
    containment <= s.containment,
    connection <= s.connection,
    roleName <= s.roleName
  )
}

helper def : createMethods(s : PLM! CObject) : Sequence(PLM2! Method) =
  s._l_.feature -> iterate (i; ret : Set(PLM2! Method) = Sequence{} | if (i -> oclType() -> conformsTo(PLM2! Method))
    then ret -> including (thisModule.createMethod(i))
    else ret
  endif
):

helper def : createAttributes(s : PLM! CObject) : Sequence(PLM2! Attribute) =
  s._l_.feature -> iterate (i; ret : Set(PLM2! Attribute) = Sequence{} | if (i -> oclType() -> conformsTo(PLM2! Attribute))
    then ret -> including (thisModule.createAttribute(i))
    else ret
  endif
):

lazy rule createMethod {
  from
  s : PLM! Method
  to
  t : PLM2! Method (s

unique lazy rule createAttribute {
  from
    s : PLM! Attribute
  to
    t : PLM2! Attribute (  
      __xmlID__ <- s.__xmlID__,  
      name <- s.name,  
      body <- s.body  
    )
}

Listing C.1: Struct_View2SUM transformation
Appendix D

Context_View2SUM

This transformation contains the code to merge the context view with the SUM model. This results into the temporary SUM file, which will be merged with the old SUM model using the Merge_SUM transformation afterwards.

```ocaml
module Extended_Context2SUM;
create OUT : PLM2 from IN : PLM;

helper def : system : OclAny = PLM!"O0. System"->allInstancesFrom(‘IN’) -> first();
-- Get all the relationship connections that are connected to the system.
helper def : relf : OclAny = PLM!"O0. RelationshipFrom"->allInstancesFrom(‘IN’) -> select( i | "<thisModule.conn(i))
helper def : relt : OclAny = PLM!"O0. RelationshipTo" -> allInstancesFrom(‘IN’) -> select( i | "<thisModule.conn(i))

helper def : conn (c : PLM!Connection) : Boolean = (c.getAllParticipations() -> exists ( j | "<thisModule.part(j))
helper def : part (p : PLM!Participation) : Boolean = (thisModule.system = p.destination);

rule System {
    from s : PLM!"O0. System"
    to
    t : PLM2!"O0. System" ("_l_.xmlID_" = s._l_.xmlID_,
    name = s.name,
    "_l_.name" = s._l_.name,
    "_l_.feature" = thisModule.createMethods(s),
    "_l_.potency" = s._l_.potency)
    do
    thisModule.system = t;
}

rule SubSystem {
    from s : PLM!"O0. SubSystem"
    to
```
```java
t : PLM2!"O0. SubSystem" (  
    ".xmiID"  <- s."xmiID",  
    name  <- s.name,  
    ".name"  <- s."name",  
    ".feature"  <- thisModule.createMethods(s),  
    ".potency"  <- s."potency")
}

rule Actor {
  from  
  s : PLM!"O0. Actor"  
  to  
  t : PLM2!"O0. Actor" (  
    ".xmiID"  <- s."xmiID",  
    name  <- s.name,  
    ".name"  <- s."name",  
    type  <- s.type,  
    ".potency"  <- s."potency",  
    ".feature"  <- thisModule.createMethods(s))
}

rule RelationshipFrom {  
  from  
  s : PLM!"O0. RelationshipFrom" ( not(thisModule.relf -> includes(s)))  
  to  
  t : PLM2!"O0. RelationshipFrom" (  
    ".xmiID"  <- s."xmiID",  
    ".name"  <- s."name",  
    description  <- s.description,  
    ".participation"  <- (s.getAllParticipations() -> collect(e | thisModule.createParticipations(e))),  
    ".feature"  <- thisModule.createMethods(s))
}

rule RelationshipTo {  
  from  
  s : PLM!"O0. RelationshipTo" ( not(thisModule.relt -> includes(s)))  
  to  
  t : PLM2!"O0. RelationshipTo" (  
    ".xmiID"  <- s."xmiID",  
    ".name"  <- s."name",  
    description  <- s.description,  
    ".participation"  <- (s.getAllParticipations() -> collect(e | thisModule.createParticipations(e))),  
    ".feature"  <- thisModule.createMethods(s))
}

rule RelationshipInternal {  
  from  
  s : PLM!"O0. RelationshipInternal"  
  to  
  t : PLM2!"O0. RelationshipInternal" (  
    ".xmiID"  <- s."xmiID",  
    ".name"  <- s."name",  
    description  <- s.description,  
    ".participation"  <- (s.getAllParticipations() -> collect(e | thisModule.createParticipations(e))),  
    ".feature"  <- thisModule.createMethods(s))
}

rule RelationshipExternal {  
  from
```
s : PLM!"O0. RelationshipExternal"

to
t : PLM2!"O0. RelationshipExternal" ( 106
"_l__xmlID__" ← s._l__xmlID__,
"_l__name" ← s._l__name,

description ← s.description,
"_l__participation" ← (s.getAllParticipations() → collect (e | thisModule.
  → createParticipations(e))))
"_l__feature" ← thisModule.createMethods(s)
)
}

rule ConsistsOf {  
from  
s : PLM!"O0. ConsistsOf"

to
}

lazy rule createParticipations {  
from  
s : PLM! Participation

to
}

helper def : createMethods(s : PLM! Clobject) : Sequence(PLM2!Method) = 146
  s._l__feature → iterate (i; ret: Set(PLM2!Method) = Sequence{} |  
  if (i →oclType() → conformsTo(PLM! Method))  
  then ret → including (thisModule, createMethod(i))  
  else ret  
  end)

helper def : createAttributes(s : PLM! Clobject) : Sequence(PLM2! Attribute) = 156
  s._l__feature → iterate (i; ret: Set(PLM2! Attribute) = Sequence{} |  
  if (i →oclType() → conformsTo(PLM! Attribute))  
  then ret → including (thisModule, createAttribute(i))  
  else ret  
  end)

unique lazy rule createMethod {  
from  
s : PLM! Method

to
}
unique lazy rule createAttribute {
    from
        s : PLM! Attribute
    to
        t : PLM2! Attribute (  
        _xmiID_ <- s._xmiID_,
        name <- s.name,
        value <- s.value,
        mutability <- s.mutability,
        durability <- s.durability  
    )
}

Listing D.1: Context_View2SUM transformation
Appendix E

Merge_SUM

This transformation contains the code to merge the temporary SUM model with the old SUM model. This results in the new SUM model containing all the changes that have been made.
i : PLM0! Classification (                      
    instance <- t,                                
    type <- type                                 
  )
}
do{
  connection <- thisModule.transformConnection(s.getEigenConnections());
  instModel.content <- instModel.content->append(t);
  instModel.content <- instModel.content->append(i);
  instModel.content <- instModel.getConnections()->including(connection)->flatten();
}

helper def : transformConnection(s : Set(PLM1!Connection)) : Set(PLM0!Connection) =
  s->iterate(i : ret : Set(PLM0!Connection) = Set{} | ret -> including(thisModule.conn(i)));

unique lazy rule conn {
  from s : PLM1! Connection
  using {
    typeModelInd : Integer = s.getDirectTypes()->first().getLevel().getLevel() + 1;
    instModel : PLM0! Level = PLM0!DeepModel->allInstancesFrom("OUT0")->first().content->at(
      typeModelInd);
    instModel.content <- instModel.getConnections()->including(thisModule.conn(i))->flatten();
    type : PLM0! Connection = PLM0!Connection->allInstancesFrom("OUT0")->select(e|e.xmiID==s.
      getDirectTypes()->first().xmiID->first());
  } to t : PLM0! Connection (                      
    __xmiID__ <- s.__xmiID__,                   
    name <- s.name,                            
    participation <- thisModule.transformParticipation(s.participation),
    feature <- thisModule.transformFeature(s.feature) )
},
i : PLM0! Classification (                      
    instance <- t,                                
    type <- type                                 
  )
do{
  instModel.content <- instModel.content->append(i);
}
-- need classification

helper def : transformParticipation(s : Set(PLM1!Participation)) : Set(PLM0!Participation) =
  s->iterate(i : ret : Set(PLM0!Participation) = Set{} | ret -> including(thisModule.part(i)));

unique lazy rule part {
  from s : PLM1! Participation
  to t : PLM0! Participation (                      
    __xmiID__ <- s.__xmiID__,                   
    roleName <- s.roleName,                    
    destination <- PLM1!Entity->allInstancesFrom("IN1")->union(PLM1!Entity->
      allInstancesFrom("IN2"))->flatten() 
    ->select(i | thisModule.toTransform(i) and i.__xmiID__==s.
      getDirectTypes()->first().xmiID->first()),
    navigable <- s.navigable
  )
}

helper def : transformFeature(s : Set(PLM1!Feature)) : Set(PLM0!Feature) =
  s->iterate(i : ret : Set(PLM0!Feature) = Set{} | if (i->oclIsTypeOf(PLM1!Attribute))
    then ret -> including(thisModule.attr(i))
else if (i->oclIsTypeOf(PLM1!Method))
    then ret -> including(thisModule.meth(i))
else ret
endif
endif

unique lazy rule attr {from
    s : PLM1!Attribute
to
    t : PLM0!Attribute (__xmlID__ <- s.__xmlID__,
    name <- s.name,
    value <- s.value
    )
}

unique lazy rule meth {from
    s : PLM1!Method
to
    t : PLM0!Method (__xmlID__ <- s.__xmlID__,
    name <- s.name,
    body <- s.body
    )
}
lazy rule Inst {
    from
    s : PLM1!Entity , type:PLM1!Entity
    using {
        typeModelInd : Integer = type.getLevel().getLevel() + 1;
        instModelInd : Integer = typeModelInd + 1;
        typeModel : PLM0!Level = PLM0!DeepModel->allInstancesFrom('OUT0')->first().content->at(
            typeModelInd);
        instModel : PLM0!Level = PLM0!DeepModel->allInstancesFrom('OUT0')->first().content->at(
            instModelInd);
    }
    to
    t : PLM0!Entity (name <- s.name,
        __xmlID__ <- s.__xmlID__
    ), i : PLM0!Classification (instance <- t,
        type <- type
    )
    do {
        instModel.content <- instModel.content->append(t);
        instModel.content <- instModel.content->append(i);
Appendix F

Helper_SUM2View

This transformation contains the code to generate the helper view, which is used to connect the sub-systems with the components.

```xml
-- @nsURI __TRACE=http://daisee/trace
-- @nsURI SUM=http://melanee.org/PLM
-- @nsURI Helper=http://melanee.org/PLM
-- @author p.lanting

module Extended_SUM2Helper;

create OUT : Helper from IN : SUM, parameters : Parameters;

rule SubSystem {
  from
  s : SUM!"00.SubSystem"
  to
  t : Helper!"00.SubSystem" ( 
    "_l_.xmlID_" <= s._l_.xmlID_,
    name <= s.name,
    "_l_.name_" <= s._l_.name,
    "_l_.feature_" <= thisModule.createMethods(s),
    "_l_.potency_" <= s._l_.potency
  )
}

rule Component {
  from
  s : SUM!"00.Component"
  to
  t : Helper!"00.Component" ( 
    "_l_.xmlID_" <= s._l_.xmlID_,
    name <= s.name,
    "_l_.name_" <= s._l_.name,
    "_l_.feature_" <= thisModule.createMethods(s),
    "_l_.potency_" <= s._l_.potency
  )
}

rule Includes {
  from
  s : SUM!"00.Includes"
  to
  t : Helper!"00.Includes" ( 
    "_l_.xmlID_" <= s._l_.xmlID_,
    "_l_.name_" <= s._l_.name,
    "_l_.participation_" <= (s.getAllParticipations() -> collect (e | thisModule.
```
lazy rule createParticipations {
  from
  s : SLM! Participation
  to
  t : Helper! Participation
  \n  _xmIID.. <= s._xmIID..
  destination <= s.destination,
  lower <= s.lower,
  upper <= s.upper,
  navigable <= s.navigable,
  containment <= s.containment,
  connection <= s.connection,
  roleName <= s.roleName
}

helper def : createMethods(s : SUM! Clabject) : Sequence(Helper! Method) =
  s._l_.feature -> iterate (i; ret: Set(Helper! Method) = Sequence{} |
    if (i -> oclType() -> conformsTo(SUM! Method))
      then ret -> including(thisModule.createMethod(i))
    else ret
eendif
)

helper def : createAttributes(s : SUM! Clabject) : Sequence(Helper! Attribute) =
  s._l_.feature -> iterate (i; ret: Set(Helper! Attribute) = Sequence{} |
    if (i -> oclType() -> conformsTo(SUM! Attribute))
      then ret -> including(thisModule.createAttribute(i))
    else ret
eendif
)

unique lazy rule createMethod {
  from
  s : SLM! Method
  to
  t : Helper! Method
  _xmIID.. <= s._xmIID..
  name <= s.name,
  body <= s.body
}

unique lazy rule createAttribute {
  from
  s : SLM! Attribute
  to
  t : Helper! Attribute
  _xmIID.. <= s._xmIID..
  name <= s.name,
  value <= s.value,
  mutability <= s.mutability,
  durability <= s.durability
}
Appendix G

Helper_View2SUM

This transformation contains the code to merge the helper view, which is used to connect the sub-systems with the components.

```xml
module Extended_Helper2SUM;
create OUT : PLM2 from IN : PLM;

rule SubSystem { from s : PLM!"O0.SubSystem" to t : PLM2!"O0.SubSystem" {
"l_.xmlID_" <= s."l_.xmlID_",
"l_.name" <= s."l_.name",
"l_.feature" <= thisModule.createMethods(s),
"l_.potency" <= s."l_.potency"
}
}

rule Component { from s : PLM!"O0.Component" to t : PLM2!"O0.Component" {
"l_.xmlID_" <= s."l_.xmlID_",
"l_.name" <= s."l_.name",
"l_.feature" <= thisModule.createMethods(s),
"l_.potency" <= s."l_.potency"
}
}

rule Includes { from s : PLM!"O0.Includes" to t : PLM2!"O0.Includes" {
"l_.xmlID_" <= s."l_.xmlID_",
"l_.name" <= s."l_.name",
"l_.participation" <= (s.getAllParticipations() -> collect (e | thisModule.
createParticipations(e)))
}
```
lazy rule createParticipations {
  from s : PLM! Participation to t : PLM2! Participation {
    _xmiID_ <- s.xmiID_,
    roleName <- s.roleName,
    destination <- s.destination,
    lower <- s.lower,
    upper <- s.upper,
    navigable <- s.navigable,
    containment <- s.containment,
    connection <- s.connection
  }
}

helper def : createMethods(s : PLM! Clobject) : Sequence(PLM2! Method) =
  s._.feature -> iterate(i; ret: Set(PLM2! Method) = Sequence{} | if (ioclType() -> conformsTo(PLM! Method))
    then ret -> including(thisModule.createMethod(i))
  else ret
endif
);

helper def : createAttributes(s : PLM! Clobject) : Sequence(PLM2! Attribute) =
  s._.feature -> iterate(i; ret: Set(PLM2! Attribute) = Sequence{} | if (ioclType() -> conformsTo(PLM! Attribute))
    then ret -> including(thisModule.createAttribute(i))
  else ret
endif
);

unique lazy rule createMethod {
  from s : PLM! Method to t : PLM2! Method {
    _xmiID_ <- s._xmiID_,
    name <- s.name,
    body <- s.body
  }
}

unique lazy rule createAttribute {
  from s : PLM! Attribute to t : PLM2! Attribute {
    _xmiID_ <- s._xmiID_,
    name <- s.name,
    value <- s.value,
    mutability <- s.mutability,
    durability <- s.durability
  }
}

Listing G.1: Helper_View2SUM transformation
Appendix H

Dimension Configuration file

This is the dimension configuration file that is used to configure the dimensions for the extension to nAOMi.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<DimConfig:Configuration xmlns:xmi="http://www.omg.org/XMI" xmlns:DimConfig="http://swt.informatik.uni-mannheim.de/nAOMi/DimConfig">
  <dimensionContainer>
    <dimension name="Version" icon="" contentProvider="/ @contentProviderContainer/
      @contentProvider.0" contentComparator="/ @contentComparatorContainer/@contentComparator.0" />
      <dimensionElement name="latest"/>
    </dimension>
    <dimension name="Abstraction" startMinimized="false">
      <dimensionElement name="PSM"/>
    </dimension>
    <dimension name="Component" icon="icons/Component.gif" contentProvider="/ @contentProviderContainer/ @contentProvider.1" contentComparator="/ @contentComparatorContainer/@contentComparator.1">
      <dimensionElement name="CompleteOverview" defaultSelection="true"/>
    </dimension>
    <dimension name="Encapsulation"/>
    <dimensionElement name="Specification" defaultSelection="true" icon="icons/blackbox.gif"/>
    <dimensionElement name="Realization" icon="icons/whitebox.gif"/>
    <dimension name="Projection" icon=""/>
    <dimensionElement name="Structural"/>
    <dimensionElement name="Behavioural"/>
    <dimensionElement name="Contextual"/>
    <dimensionElement name="Helper"/>
    <dimension name="Granularity"/>
    <dimensionElement name="Service" icon="icons/Component.gif"/>
    <dimensionElement name="Type" icon="icons/Class.gif"/>
    <dimension name="Operation" icon="icons/Operation.gif" contentProvider="/ @contentProviderContainer/@contentProvider.2" contentComparator="/ @contentComparatorContainer/@contentComparator.1">
      <dimensionElement name="all_owned" icon=""/>
    </dimension>
   </dimensionContainer>
 <mappingContainer>
  <mapping transformation="/ @transformationContainer/@transformation.0" editor="/ @viewEditorContainer/@editor.0">
    <dimElSpec dimElementName="/ @dimensionContainer/@dimension.1/@dimensionElement.1" dimName="/ @dimensionContainer/@dimension.1"/>
 </mappingContainer>
</DimConfig:Configuration>
```
<dimElSpec dimElementName="@dimensionContainer/@dimension.4/@dimensionElement.0" dimName="/ @dimensionContainer/@dimension.4"/>
<dimElSpec dimElementName="@dimensionContainer/@dimension.5/@dimensionElement.0" dimName="/ @dimensionContainer/@dimension.5"/>
<dimElSpec dimElementName="@dimensionContainer/@dimension.6/@dimensionElement.0" dimName="/ @dimensionContainer/@dimension.6"/>
</mapping>
<dimElSpec dimElementName="/ @dimensionContainer/@dimension.1/ @dimensionElement.1" dimName="/ @dimensionContainer/@dimension.1"/>
<dimElSpec dimElementName="/ @dimensionContainer/@dimension.3/ @dimensionElement.0" dimName="/ @dimensionContainer/@dimension.3"/>
<dimElSpec dimElementName="/ @dimensionContainer/@dimension.4/ @dimensionElement.1" dimName="/ @dimensionContainer/@dimension.4"/>
<dimElSpec dimElementName="/ @dimensionContainer/@dimension.5/ @dimensionElement.0" dimName="/ @dimensionContainer/@dimension.5"/>
<dimElSpec dimElementName="/ @dimensionContainer/@dimension.6/ @dimensionElement.0" dimName="/ @dimensionContainer/@dimension.6"/>
</mapping>
<dimElSpec dimElementName="/ @dimensionContainer/@dimension.2/ @dimensionElement.2" dimName="/ @dimensionContainer/@dimension.4"/>
<dimElSpec dimElementName="/ @dimensionContainer/@dimension.4/ @dimensionElement.2" dimName="/ @dimensionContainer/@dimension.4"/>
<dimElSpec dimElementName="/ @dimensionContainer/@dimension.5/ @dimensionElement.0" dimName="/ @dimensionContainer/@dimension.5"/>
<dimElSpec dimElementName="/ @dimensionContainer/@dimension.6/ @dimensionElement.0" dimName="/ @dimensionContainer/@dimension.6"/>
</mapping>
<dimElSpec dimElementName="/ @dimensionContainer/@dimension.3/ @dimensionElement.3" dimName="/ @dimensionContainer/@dimension.4"/>
<dimElSpec dimElementName="/ @dimensionContainer/@dimension.4/ @dimensionElement.3" dimName="/ @dimensionContainer/@dimension.4"/>
<dimElSpec dimElementName="/ @dimensionContainer/@dimension.5/ @dimensionElement.0" dimName="/ @dimensionContainer/@dimension.5"/>
<dimElSpec dimElementName="/ @dimensionContainer/@dimension.6/ @dimensionElement.0" dimName="/ @dimensionContainer/@dimension.6"/>
</mapping>
</mappingContainer>
</transformationContainer>
<transformation name="Structural" plugin_id="nl.utwente.naomi.trafo.extensionView" class="nl.utwente.naomi.trafo.extensionview.util.TrafoLauncherExtensionView" id="Structural">
<sumToView launcher="Regular_VM transformationFile="/Transformations/Structural/struct.Sum2VIEW.asm" />
</sumToView>
</transformation>
<transformation name="Behavioural" plugin_id="nl.utwente.naomi.trafo.extensionView" class="nl.utwente.naomi.trafo.extensionview.util.TrafoLauncherExtensionView" id="Behavioural">
<sumToView launcher="Regular_VM transformationFile="/Transformations/Behavioural/behaviour.Sum2VIEW.asm" />
</transformation>
Listing H.1: Dimension configuration file