Dissertation

Software Architecture Modeling by Reuse, Composition and Customization

Ivano Malavolta

March 2012

Advisor
Dr. Henry Muccini

PhD Program Supervisor
Prof. Stefania Costantini

© Ivano Malavolta, 2012. All rights reserved
ABSTRACT

While developing a complex system, it is of paramount importance to correctly and clearly specify its software architecture. Architecture Description Languages (ADLs) are the means to define the software architecture of a system. ADLs are strongly related to stakeholder concerns: they must capture all design decisions fundamental for systems stakeholders. From the earliest work in software architecture, the usefulness of expressing software architectures in terms of multiple views is well recognized. Architecture views represent distinct aspects of the system of interest and are governed by viewpoints which define the conventions for their construction, interpretation and use to frame specific system concerns. Most practising software architects operate within an architecture framework which is a coordinated set of viewpoints, models and notations prescribed for them. As a matter of fact, stakeholders concerns vary tremendously, depending on the project nature, on the domain of the system to be realized, etc. So, even if current architecture frameworks are defined to varying degrees of rigour and offer varying levels of tool support, finding the right architecture framework that allows to address the various system concerns is both a risky and difficult activity. Therefore, an effective way to define and combine architectural elements into a suitable framework for effectively create architecture descriptions is still missing.

In this dissertation, I propose an infrastructure for modeling the architecture of a software system by adapting existing architectural languages, viewpoints and frameworks to domain- and organization-specific features. Under this perspective, the proposed infrastructure allows architects to set up customized architectural frameworks by: (i) defining and choosing a set of viewpoints that adequately fit with the domain and features of the system being developed, (ii) automatically adapting existing architecture description languages to project-specific concerns, (iii) keeping architectural views within the framework synchronized, (iv) enabling consistency and completeness checks based on defined correspondences and rules among architectural elements. The proposed approach builds upon the conceptual foundations of ISO/IEC/IEEE 42010 for architecture description and it is generic with respect to the used architectural elements (i.e., views, viewpoints, languages, stakeholder’s concerns, etc.).

The impact of the proposed approach is three-fold: (i) a novel approach is presented for architecting by reusing, composing and customizing existent architectural elements, (ii) a new composition mechanism is presented for extending architectural languages in a controlled fashion, (iii) a new mechanism for keeping architectural views in a consistent state is provided.

The proposed approach is realized through a combination of model transformations, weaving, and megamodeling techniques. The approach has been put in practice in different scenarios and has been evaluated in the context of a real complex system.
"If you think good architecture is expensive, try bad architecture"
Brian Foote and Joseph Yode
# TABLE OF CONTENTS

Abstract i  
Table of Contents v  
List of Figures ix  
List of Tables xiii

## 1 Introduction

1.1 Software Architecture Modeling 2  
1.2 Motivation 3  
1.3 Problem Statement: Research Questions 5  
1.4 The Proposed Solution 6  
   1.4.1 Cross-view Consistency 7  
   1.4.2 Languages Extension and Customization 8  
   1.4.3 Wrap up 8  
1.5 Contributions 9  
1.6 Structure of this Dissertation 11

## 2 Background

2.1 Software Architecture: Concepts and Terminology 13  
   2.1.1 Overview of ISO/IEC/IEEE 42010 14  
   2.1.2 Architecture Description, Stakeholders and Concerns 16  
   2.1.3 Architecture Views and Viewpoints 17  
   2.1.4 Architectural Correspondences 20  
   2.1.5 Architecture Decisions and Rationale 21  
   2.1.6 Architecture Description Languages 23  
   2.1.7 Architecture Frameworks 26  
2.2 Model-driven Engineering Concepts 28  
   2.2.1 The metamodeling stack 29  
   2.2.2 Model Transformations 31  
   2.2.3 Model Weaving 33  
   2.2.4 Megamodeling 35

## 3 Solution

3.1 MEGAF: an infrastructure for realizing architecture frameworks 41  
   3.1.1 Conceptual overview of MEGAF 43  
   3.1.2 The SBSCS Example 47
# TABLE OF CONTENTS

## 3.2 DUALLY: a solution for providing architecture description languages interoperability
- 3.2.1 The DUALLY model transformations engine ............................................. 49
- 3.2.2 The \(A_0\) pivot language ........................................................................ 54
- 3.2.3 The ACC Case Study ............................................................................... 58

## 3.3 BYADL: a solution for extending and customizing architectural languages
- 3.3.1 Abstract syntax of the extended ADL ......................................................... 67
- 3.3.2 Concrete syntaxes of the extended ADL ....................................................... 69
- 3.3.3 Semantics of the extended ADL .................................................................. 69
- 3.3.4 Model migrators ....................................................................................... 70
- 3.3.5 The IECS Case Study ............................................................................... 72

## 4 Realization of the solution
- 4.1 Technological overview ............................................................................... 80
- 4.2 Importing ADLs and other architectural artifacts into MEGA F
  - 4.2.1 Obtaining the metamodel .................................................................... 83
  - 4.2.2 Tagging a metamodel .......................................................................... 83
  - 4.2.3 Providing semantics to a metamodel ...................................................... 85
- 4.3 Realization of MEGA F ............................................................................... 86
- 4.4 Realization of DUALLY
  - 4.4.1 Realizing the \(A_0\) Pivot Language ......................................................... 89
  - 4.4.2 Weaving ADLs to \(A_0\) ......................................................................... 93
  - 4.4.3 Automatic Transformation among Architecture Models ....................... 98
- 4.5 Realization of BYADL .............................................................................. 105
  - 4.5.1 Realization of the BYADL composition mechanism .............................. 106
  - 4.5.2 Support for concrete syntaxes generation in BYADL .............................. 110
  - 4.5.3 Support for providing semantics to a composed ADL in BYADL ............. 111
  - 4.5.4 Realization of the BYADL model migrators .......................................... 113

## 5 The BusOnAir Case Study
- 5.1 The BusOnAir Case Study .......................................................................... 119
- 5.2 Architecture Framework
  - 5.2.1 System Stakeholders and Concerns ......................................................... 122
  - 5.2.2 Architecture Viewpoints ...................................................................... 124
  - 5.2.3 Used Architecture Description Languages ............................................ 125
  - 5.2.4 Correspondence Rules ........................................................................... 137
- 5.3 Extension of DiaSpec ................................................................................... 140
- 5.4 Architecture Description
  - 5.4.1 System View .......................................................................................... 147
  - 5.4.2 Behaviour View .................................................................................... 153
  - 5.4.3 Web Services View .............................................................................. 157
- 5.5 Reflection ...................................................................................................... 159

## 6 Related Work
- 6.1 SA-specific Approaches ............................................................................. 167
- 6.2 Generic MDE Approaches ......................................................................... 172
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 Conclusions</td>
<td>175</td>
</tr>
<tr>
<td>7.1 Future Research Directions</td>
<td>179</td>
</tr>
<tr>
<td>References</td>
<td>185</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

1.1 Overview of the proposed approach ............................................ 6
1.2 Relationship between MEGAF, DUALLY and BYADL .................... 8
1.3 Structure of this dissertation .................................................. 11

2.1 Context of an Architecture description (taken from [77]) ............ 16
2.2 Architecture Description of an Adaptive Cruise Controller ............ 17
2.3 Architectural Views and Viewpoints (taken from [77]) ................. 18
2.4 Example of Architectural Views and Viewpoints ......................... 20
2.5 Architectural Correspondence and Correspondence Rule (taken from [77]) 20
2.6 Architectural Design Decision and Rationale (taken from [77]) ....... 22
2.7 Architecture Description Language in the 42010 standard (taken from [77]) 23
2.8 Example of architectural model in Darwin ................................ 25
2.9 Example of architectural model in AADL ................................ 25
2.10 Content model of an ISO/IEC/IEEE 42010 framework (taken from [77]) 27
2.11 The 4+1 view model .......................................................... 28
2.12 The 4-level metamodeling stack ............................................. 30
2.13 An overview of model transformation ...................................... 32
2.14 Weaving model ............................................................... 34
2.15 Basic Megamodeling Conceptual Framework: modeling artifacts ... 36
2.16 Basic Megamodeling Conceptual Framework: relationships ......... 37
2.17 An example of megamodel with navigation and traceability links ... 37

3.1 Conceptual overview of the proposed approach ......................... 39
3.2 Possible usage scenario of the proposed approaches .................... 40
3.3 Overview of MEGAF ......................................................... 42
3.4 Extract of GMM4SA, Megametamodel for Software Architectures ... 44
3.5 Megamodel for the running example SBSCS ............................... 47
3.6 Architecture models interoperability: generic process (a) and through A0 (b) .............................................................. 50
3.7 DUALLY High Level Conceptual View ..................................... 51
3.8 Model transformations generation in DUALLY ........................... 52
3.9 Selection of the elements of A0 ............................................ 56
3.10 Interoperability via a: a) rich A0, b) minimalistic A0. ................ 56
3.11 A0 as a hierarchy of metamodels .......................................... 58
3.12 ACC modeling with DUALLY ............................................. 59
3.13 SaveCCM model of the ACC system .................................... 60
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.14</td>
<td>$A_0$ model of the ACC system</td>
<td>61</td>
</tr>
<tr>
<td>3.15</td>
<td>Darwin model of the ACC system</td>
<td>61</td>
</tr>
<tr>
<td>3.16</td>
<td>UML model of the ACC system</td>
<td>62</td>
</tr>
<tr>
<td>3.17</td>
<td>Revised Darwin model of the ACC system</td>
<td>63</td>
</tr>
<tr>
<td>3.18</td>
<td>Revised Darwin model of the ACC system</td>
<td>63</td>
</tr>
<tr>
<td>3.19</td>
<td>Revised SaveCCM model of the ACC system</td>
<td>64</td>
</tr>
<tr>
<td>3.20</td>
<td>Revised UML$_{cc}$ model of the ACC system</td>
<td>64</td>
</tr>
<tr>
<td>3.21</td>
<td>Behaviour of the ACC Application component as UML state machines</td>
<td>65</td>
</tr>
<tr>
<td>3.22</td>
<td>Overview of BYADL</td>
<td>66</td>
</tr>
<tr>
<td>3.23</td>
<td>BYADL extended ADL generation</td>
<td>67</td>
</tr>
<tr>
<td>3.24</td>
<td>Generation of the abstract syntax of the extended ADL</td>
<td>68</td>
</tr>
<tr>
<td>3.25</td>
<td>Generation of the abstract syntax of the extended ADL</td>
<td>70</td>
</tr>
<tr>
<td>3.26</td>
<td>$Composed2SingleMigrator$ model migrator in BYADL</td>
<td>71</td>
</tr>
<tr>
<td>3.27</td>
<td>$Singe2ComposedMigrator$ model migrator in BYADL</td>
<td>71</td>
</tr>
<tr>
<td>3.28</td>
<td>ADL composition scenarios for the IECS system</td>
<td>72</td>
</tr>
<tr>
<td>3.29</td>
<td>DarwinFT: Extending Darwin with Fault Tolerance</td>
<td>73</td>
</tr>
<tr>
<td>3.30</td>
<td>DarwinFT+BPMN: DarwinFT &amp; Development process in BPMN</td>
<td>74</td>
</tr>
<tr>
<td>3.31</td>
<td>(DarwinFT+BPMN)$_{cc}$: Darwin customization</td>
<td>75</td>
</tr>
<tr>
<td>3.32</td>
<td>IECS tree-based and textual models in BYADL</td>
<td>76</td>
</tr>
<tr>
<td>3.33</td>
<td>IECS graphical model in BYADL</td>
<td>77</td>
</tr>
<tr>
<td>4.1</td>
<td>Software architecture of the proposed solution</td>
<td>79</td>
</tr>
<tr>
<td>4.2</td>
<td>The concept of Tag in the MEGAF metamegamodel</td>
<td>84</td>
</tr>
<tr>
<td>4.3</td>
<td>The SBSCS example in the MEGAF prototype</td>
<td>87</td>
</tr>
<tr>
<td>4.4</td>
<td>$A_0$ metamodel</td>
<td>89</td>
</tr>
<tr>
<td>4.5</td>
<td>Graphical interface of DULLY</td>
<td>94</td>
</tr>
<tr>
<td>4.6</td>
<td>DULLY weaving metamodel</td>
<td>96</td>
</tr>
<tr>
<td>4.7</td>
<td>Generative architecture of DULLY</td>
<td>99</td>
</tr>
<tr>
<td>4.8</td>
<td>Weaving features in DULLY</td>
<td>100</td>
</tr>
<tr>
<td>4.9</td>
<td>Basic Mechanism to manage the loss of information between architecture</td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>models.</td>
<td></td>
</tr>
<tr>
<td>4.10</td>
<td>Lost-in-translation metamodel</td>
<td>104</td>
</tr>
<tr>
<td>4.11</td>
<td>Definition of BYADL metamodel composition operators</td>
<td>106</td>
</tr>
<tr>
<td>4.12</td>
<td>BYADL composition mechanism</td>
<td>109</td>
</tr>
<tr>
<td>4.13</td>
<td>BYADL composition mechanism</td>
<td>112</td>
</tr>
<tr>
<td>4.14</td>
<td>BYADL model migrators generation</td>
<td>113</td>
</tr>
<tr>
<td>4.15</td>
<td>Sample generated Trace model</td>
<td>115</td>
</tr>
<tr>
<td>5.1</td>
<td>Workflow followed in the BusOnAir case study</td>
<td>118</td>
</tr>
<tr>
<td>5.2</td>
<td>Overview of the BusOnAir information system</td>
<td>119</td>
</tr>
<tr>
<td>5.3</td>
<td>BusOnAir Stakeholders and Concerns</td>
<td>122</td>
</tr>
<tr>
<td>5.4</td>
<td>BusOnAir Stakeholders and Concerns</td>
<td>123</td>
</tr>
<tr>
<td>5.5</td>
<td>BusOnAir Viewpoints and Model Kinds</td>
<td>125</td>
</tr>
<tr>
<td>5.6</td>
<td>Architecture Description Languages for BusOnAir</td>
<td>126</td>
</tr>
<tr>
<td>5.7</td>
<td>Extended $A_0$ with behavioural concepts</td>
<td>126</td>
</tr>
</tbody>
</table>
5.8 Sense-Compute-Control application .......................... 128
5.9 DUALLY weaving model between DiaSpec and $A_0$ .......... 130
5.10 UML$_{cc}$ profile .................................................. 131
5.11 DUALLY weaving model between UML$_{cc}$ and $A_0$ ....... 132
5.12 The FSP metamodel .............................................. 133
5.13 DUALLY weaving model between DarwinFSP and $A_0$ .... 134
5.14 The REST metamodel ............................................ 136
5.15 The componentMM metamodel ................................ 140
5.16 Composition of DiaSpec and ComponentMM ................ 141
5.17 Architecture views for the BusOnAir architecture description .... 143
5.18 Workflow for architecting the BusOnAir system ............. 144
5.19 Workflow for architecting the BusOnAir system (detail on the used models) 146
5.20 Simulation of the Bus system in Diasuite ..................... 149
5.21 Bus System model in DiaSpecComp ............................. 150
5.22 BusOnAir model in DiaSpecComp ............................... 151
5.23 BusOnAir model in UML$_{cc}$ .................................. 152
5.24 Behaviour of the TripManager component in UML$_{cc}$ .... 154
5.25 Behaviour of the TripManager component in FSP (graphical representation) 156
5.26 Animation of the TripManager behaviour in FSP ............. 157
5.27 BusOnAir REST services model in RESTLANG ............... 158
7.1 Eclipse Zest Screenshot .......................................... 181
7.2 Concurrent changes in DUALLY ................................. 182
LIST OF TABLES

1.1 Usefulness of architectural language features in past projects . . . . . 4
1.2 Usefulness of architectural language features in future projects . . . . 4
5.1 Comparison between point-to-point and DUALLY . . . . . . . . . . . . 164
While developing a complex system, it is fundamental to correctly and clearly specify its software architecture (SA) [142]. Architecture Description Languages (ADLs) are the means to define the software architecture of a system. ADLs are strongly related to stakeholder concerns: they must capture all design decisions fundamental for systems stakeholders. From the earliest work in software architecture, it has been a fundamental tenet of the field that architectures are best expressed in terms of multiple views [122], each view representing a distinct aspect of the system of interest. Academic research and existing architecture description languages (ADLs) [110] have focused predominantly on the structural view (i.e. components and connectors) and on behaviour at the architectural level. However, they have offered limited support to address the needs of stakeholders with different concerns such as data management, safety, security, reliability, design decisions, and so on [77, 34, 142].

Most practising software architects operate within an architecture framework which is a coordinated set of viewpoints, models and notations prescribed for them [77]. As a matter of fact, stakeholders concerns vary a lot, depending on the project nature, on the domain of the system to be realized, and on several other aspects. So, even if current architecture frameworks are defined to varying degrees of rigour and offer varying levels of tool support, they tend to be too narrow with respect to stakeholders concerns [72]. As a consequence, it is difficult to adequately support all system concerns by employing predefined architectural elements like modeling languages, views, and frameworks. Therefore, an effective way to define and combine architectural elements into a suitable framework for effectively create architecture descriptions is still missing. As analysed in [44], once an organization has defined a framework for the architectural elements to be used within its domain, the organization can more easily capitalize investments in evaluation, training, and automated tools.

In this dissertation, I propose an infrastructure for modeling the architecture of a software system by adapting existing architectural languages, viewpoints and frameworks to domain- and organization-specific features. Such an infrastructure provides the basis for defining, reusing, and sharing architectural elements across projects (and the community) and for creating architecture descriptions that better fit with the system concerns. Under this perspective, the proposed infrastructure allows architects to set up customized architectural frameworks that conform to the ISO/IEC/IEEE 42010:2011, Sys-
tems and software engineering Architecture description [77] standard for architecture description and that better fit with the system under development. The ISO/IEC/IEEE 42010 standard will be described in Section 2.1. The proposed approach is realized through a combination of model transformations, weaving, and megamodelling techniques.

In the remainder of this chapter, I start by explaining the research context of this work by providing a very brief overview about software architecture modeling (basic notions of software architecture, architecture description language, architectural view, viewpoint, framework are provided here). Next, in Section 1.2 I will give the main issues that motivate the work, and in Section 1.3 the main research questions that this work aims to address are provided. This introduction concludes with a summary of the contributions of this work (Section 1.5, and an overview of the dissertation 1.6.

1.1 SOFTWARE ARCHITECTURE MODELING

During the development of a complex system, it is fundamental to correctly and clearly specify its software architecture (SA) [142]. Clearly, a precise definition of the software architecture of a system provides considerable benefits for the system’s stakeholders. For example, the description of an SA of the system allows an early analysis of whether the system can meet its requirements, it may be used as a focus of discussion by system stakeholders, it allows for reasoning on the system from the very early stages of its development life-cycle, it may be used as a high-level design blueprint to be used during system development, testing, maintenance, and so on.

One of the main results of a recent survey on the architecting activities in industry [105] is that nowadays the majority of architects express the software architecture of a system in terms of multiple architectural views, each view addressing a given set of system concerns which are held by various stakeholders [122]. A view expresses the architecture of the system-of-interest in accordance with a specific architecture viewpoint. It establishes the conventions for constructing, interpreting and analysing an architecture view. Such conventions may include languages, design rules, modelling methods, analysis techniques and other auxiliary operations on views. The use of multiple views has become standard practice in industry [94, 75, 142], yet academic research and existing architecture description languages have focused predominantly on the structural view (i.e. components and connectors) and sometimes on behaviour at the architectural level, and have offered limited support to address the needs of stakeholders with different concerns such as data management, safety, security, reliability and so on\(^1\).

A specific view (or a set of views) of the system can be defined by means of an Architecture Description Language (ADL) [110]. Fundamentally, an ADL is a kind of domain specific language for specifying software architectures. Early ADLs proposed during the 1990s were generic and had ADLs designers had the main purpose to define the “ideal”

\(^1\)See [77] for extensive discussion of the range of architecturally-relevant system concerns.
ADL [60]: a language able to describe any feature and element constituting a software architecture. These languages mainly focused on system components, connectors and their overall interconnection [122, 60]. Then ADLs evolved into a new generation of notations, each one dealing with more specific features, such as configuration management, language extensibility, and product-line modelling support.

While developing a complex system, ADLs, viewpoints, views and notations are coordinated in the context of an architecture framework [77]. It is defined as a coordinated set of viewpoints, models and notations prescribed for them. An ADL can be considered as a mini architecture framework [77]: it allows to frame concerns, it is suited for particular stakeholders, it may be divided into one or more model kinds, and it is possibly organized into viewpoints. There are many architecture frameworks in use today (like Zachman, TOGAF, GERAM, and DODAF), each with different levels of abstraction, various degrees of rigour and offering varying levels tool support.

A detailed description of all the architectural concepts described so far is provided in Section 2.1.

1.2 Motivation

As anticipated at the beginning of this chapter, stakeholders may have many different concerns with respect to the system being developed. Stakeholders concerns may vary depending on the nature of the system being developed, project-specific constraints, or the domain of the system being developed (e.g., automotive, aerospace, finance, health-care) [77]. As experimentally emerged from a recent survey we conducted on the usage of architecture description languages in industry [105], it is common that industrial architects try to manage the complexity of considering such a high number of variable (and usually project-specific) concerns by:

1. extending (where possible) the used architecture description languages so that they consider the specificities of the identified stakeholders concerns;

2. describing the architecture of the system by means of multiple views.

More in detail, Table 1.1 and Table 1.2 are an excerpt of a result in [105]; it shows how participants ranked which architectural language features they think are useful for their past projects, and what features would be useful for future projects.

This is an indication that software architects (i) do extend (where possible) architectural languages when architecting in practice, and (ii) do use multiple views.

---


3The ranking is between -2 (entirely useless) to +2 (entirely useful)
Table 1.1: Usefulness of architectural language features in past projects

<table>
<thead>
<tr>
<th>Feature</th>
<th>-2</th>
<th>-1</th>
<th>0</th>
<th>+1</th>
<th>+2</th>
<th>No exp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extensibility</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>13</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>Customization</td>
<td>1</td>
<td>4</td>
<td>11</td>
<td>12</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Support for multiple architectural views</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>11</td>
<td>18</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 1.2: Usefulness of architectural language features in future projects

<table>
<thead>
<tr>
<th>Feature</th>
<th>-2</th>
<th>-1</th>
<th>0</th>
<th>+1</th>
<th>+2</th>
<th>No exp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extensibility</td>
<td>1</td>
<td>2</td>
<td>10</td>
<td>12</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>Customization</td>
<td>1</td>
<td>2</td>
<td>11</td>
<td>12</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>Support for multiple architectural views</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>27</td>
<td>1</td>
</tr>
</tbody>
</table>

Those findings are in line with the broader view of SA that is accepted today, which goes far beyond the traditional perception of an SA as a set of constituting elements and looks at multiple stakeholder’s concerns and their design decisions [77, 93, 142]. Based on this broader view of SA, what is evident nowadays is that an ideal and general purpose architectural language cannot exist, and architectural languages must be extensible so to be extended and customized according to stakeholder’s concerns. Moreover, as shown in the survey we conducted, in most of the cases they had to extend the used architectural language to cope with domain-specific concepts, constraints, or analysis needs.

Also, about 85% of the respondents of our survey on architectural languages do use multiple views for architectural description. This is in line with the perception that software architectures are best expressed in terms of multiple views, each view representing a distinct aspect of the system of interest. Even if the ubiquitous advice on software architecture is to build multiple views of the system (since they help by focusing attention on one aspect and creating a single view would create a muddle of details that defeats the purpose of having a model), in the current state of the art view consistency is one of the harder problems in software architecture [47].

One consequence of the tenet of using multiple views is a growing body of viewpoints that have become available, such as [94, 131, 18, 17, 92]. Another consequence is the rise of architecture frameworks as coordinated sets of viewpoints. Most practising software architects must operate within an architecture framework prescribed for them by their organization or client (current frameworks include Zachman, TOGAF, GERAM, and DODAF). There are many such frameworks in use today, defined to varying degrees of rigour and offering varying levels of automated tool support. However, these frameworks tend to be closed—as a result, (i) it is difficult to re-use viewpoints and concerns
for defining new frameworks to be used in different organizations or domains; and (ii) it is difficult to keep consistency rules among viewpoints once forever, since such rules are not re-usable as the main artifacts themselves. Therefore, an effective way of managing, storing, retrieving, and combining architectural elements, in order to build a coordinated set of viewpoints (i.e., a framework) within a specific domain of application or community of stakeholders is still missing.

With the aim of taking a step towards the solution of these current issues, the goal of this dissertation is to provide an infrastructure for modeling the architecture of a software system by adapting and composing existing architectural languages, views and frameworks to domain- and organization-specific features.

1.3 Problem Statement: Research Questions

Architectural languages are strongly related to system’s stakeholder concerns, thus an architectural language must provide a good level of abstraction and the right constructs for addressing them. Clearly, the quality of the resulting architecture descriptions heavily depend on how the stakeholder concerns of the current system are supported. This issue depends on (i) the concepts supported by the used architectural languages and (ii) the architecture framework managing architecture languages, models, design decisions and the concerns themselves.

Goal of this dissertation is to answer the following research questions:

RQ1 Is it possible to reuse and compose already existing viewpoints, languages, design decisions in order to shape an architecture framework that better fit with the system’s stakeholder concerns?

RQ2 Is there a way to automatically keep consistent architectural views (and the models belonging to each of them)? Is it possible to reuse their commonalities and let them interoperate? Further, is there a scalable way to add (or remove) an architectural language from the set of used languages within the architectural framework?

RQ3 Focusing on a single architectural language, is it feasible to extend or adapt it in order to get it closer to either the domain, development process or stakeholder concerns of the system?

This dissertation aims to investigate on how current Model-Driven Engineering technologies help in answering the above mentioned research questions with a focus on reuse and automation.
1.4 **The Proposed Solution**

The intuition behind the approach proposed in this work is to provide support for getting architecture frameworks and architectural languages “closer” to system stakeholders concerns. More specifically, the approach promotes the reuse of existing architectural elements by making architecture frameworks and languages **adaptable and extensible** in order to effectively fit the system stakeholders concerns. Furthermore, the approach provides a solution to keep **consistent** the set of multiple views (and languages) used for modeling the system under development. Figure 1.1 an overview of the proposed approach.

![Figure 1.1: Overview of the proposed approach](image)

Basically, in this work I propose an approach for managing architecture descriptions by supporting the generation of architecture frameworks by defining architectural elements like stakeholders definitions, concerns, viewpoints, architectural languages and **composing** already existing ones. The resulting architecture framework conforms to the ISO/IEC/IEEE 42010 standard for architecture description. In this sense, it will help in the identification of:

- concerns,
- stakeholders having those concerns,
- the set of architecture viewpoints that frame those concerns,
- correspondence rules integrating those viewpoints,
- the set of architectural languages that identifies specific concerns and stakeholders.
Modern software architects may utilize the proposed approach as the basis for developing new architectural frameworks, that will be used to specify architecture descriptions and perform different kinds of analysis on them. I called the proposed approach MEGAF and it will be described in detail in Section 3.1.

1.4.1 CROSS-VIEW CONSISTENCY

Since MEGAF is based on the ISO/IEC/IEEE 42010 standard, it provides facilities for managing the description of the system in terms of multiple views and a set of mechanisms to check (and optionally enforce) the synchronization among them. Architects use multiple views for two reasons [77]: firstly, because different languages have different strengths for expressing various aspects of a system; secondly, because separation of concerns is a useful technique for managing complexity. Since cross-view consistency is at the same time one of the hardest problems in software architecture [47] and one of the most needed capability for an architectural framework [105], it needs to be treated with special care.

In MEGAF cross-view consistency is supported in two different ways:

1. **Descriptive**: architects can define a set of architectural correspondences representing dependencies between architectural elements within an architecture description. In the architecture framework, those correspondences are navigated by specific constraints; they will be used to check if subsets of architectural elements are in a consistent state; according to the ISO/IEC/IEEE 42010 standard, those constraints are called "correspondence rules". MEGAF allows architects to define correspondence rules via OCL (i.e., a standard language in model-driven engineering for defining constraints), and thus to executed them on specific architecture descriptions.

2. **Proactive**: special kinds of correspondence rules are defined between architectural languages. They have the additional feature of being proactive; that is, when an inconsistency is detected between different architectural models, a dedicated interoperability engine actively transforms the models in order to restore their consistency. The interoperability engine is called DUALLY; it is semi-automatic and provides a scalable transformation engine between multiple architectural models.

By means of this two mechanisms, cross-view consistency is ensured. It is important to note that the proactive mechanism asks for more effort from architects because the transformation engine must know a priori how to transform the various architectural concepts among the models. More details about the descriptive cross-view consistency is provided in Section 3.1.1, and the details about the DUALLY interoperability engine are given in Section 3.2.

MEGAF website: http://megaf.di.univaq.it
DUALLY website: http://dually.di.univaq.it
1.4.2 Languages Extension and Customization

As explained in Section 1.2, architectural languages must be extensible so to be extended and customized according to stakeholder’s concerns. Motivations for extending or customizing an architectural language range from adapting it to better express domain-specific constraints, to adding additional constraints, or to enabling additional analysis capabilities [105].

In MEGAF I decided to support the extension and customization of architectural languages by means of a dedicated engine. The engine is called BYADL; it allows to incrementally (and semi-automatically) build customized and customizable architectural languages starting from an already existing one. Newly obtained architectural languages are build by means of semantic extension mechanisms and are automatically made part of the architectural elements repository. Language extensions defined within BYADL are represented as conceptual models (more specifically, as metamodels), they are architectural language-independent and are part of the architectural elements repository as well.

Finally, the details about BYADL and how it is used within the MEGAF infrastructure will be provided in Section 3.3.

1.4.3 Wrap Up

Figure 1.2 provides an idea on how the MEGAF infrastructure is related to the DUALLY and BYADL engines.

Figure 1.2: Relationship between MEGAF, DUALLY and BYADL

[byADL website: http://byadl.di.univaq.it]
1.5 Contributions

Fundamentally, MEGAF can be considered as the main approach, in which architects can define, compose and share architectural elements. Within MEGAF, DUALLY is in charge of proactively maintaining architectural models consistency, and BYADL is used for customizing architectural languages depending on organization- and domain-specific concerns of the system being developed. So, in the remainder of this dissertation, we can consider MEGAF as the main infrastructure in which DUALLY and BYADL play the role of internal engines for managing specific sub-issues, that are architectural languages interoperability and extension, respectively.

The tangible benefits of using the MEGAF infrastructure for architecting are:

- software architects may define new frameworks by putting reusing and composing together views, viewpoints, languages, and correspondences among them depending on the system stakeholders concerns;

- software architects can use the architectural languages that best suit their needs by extending and customizing existing ones;

- software architects can model the system in an environment in which architectural languages interoperate and models conforming to different languages are automatically kept consistent.

From a technological point of view, this work approaches the research problem from a Model-Driven Engineering (MDE, [133]) perspective. So, an architectural language can be considered as a metamodel with its own tool, methodology and process, and an architectural model is a model conforming to the architectural language metamodel. By applying the MDE principle that models are precise artifacts that can be understood by computers and can be automatically manipulated, this work has a strong focus on automation and reuse. Background information on model-driven engineering and on the main motivations that drove me to choose MDE as technological solution are provided in Section 2.2.

1.5 Contributions

This research contributes with a novel approach for modeling the architecture of a software system by adapting existing architectural elements like architectural languages, views and frameworks to project- and system-specific concerns. More specifically, this dissertation exploits model-driven techniques to support the definition of an SA description by reusing (and customizing, if needed) already defined architectural elements and coordinating them in an architectural framework conforming to the ISO/IEC/IEEE 42010 standard. Concrete contributions of this dissertation are provided below.
• **A comprehensive approach for adapting architecture frameworks and architectural languages to project- and system- specific concerns.** More specifically, the approach promotes the reuse of existing architectural elements by making architecture frameworks and languages adaptable and extensible in order to effectively fit the system stakeholders concerns. In order to keep the proposed approach organized and cognitively manageable, it has been split into three sub-projects, each of them they embodying the contributions described in the next three points.

• **An infrastructure for realizing architecture frameworks,** which can be used to create architecture descriptions. It builds upon the conceptual foundations of ISO/IEC/IEEE 42010 for architecture description. The proposed infrastructure is realized through megamodelling techniques that offer the needed technology to: (i) create, store, and manage viewpoints, views, and concerns, (ii) define correspondences and rules among architectural elements, and (iii) perform consistency and completeness checks.

• **A solution that enables both interoperability and synchronization among architecture models.** It allows for an easy addition of new ADLs and guarantees convergent changes propagation. It allows architects to transform concepts of an architecture model into their corresponding concepts in other architecture models. The main advantages that the proposed solution for interoperability exposes can be summarized as follows: (i) it works at two abstraction levels (i.e., metamodeling and modeling), providing a clear separation between MDE experts (the technical stakeholders) and software architects (the final users); (ii) the model transformation engine is completely hidden to software architects, making the solution easier to adopt for them; (iii) it allows software architects to transform among formal ADLs and UML model-based notations and vice versa; (iv) software architects can continue to use their familiar architectural notations and tools, and can reuse existing architectural models; (v) the correspondences between two architecture description languages are defined once, and reused for each architecture model governed by them.

• **An incremental approach to extend and customize ADLs** via a set of well formalized metamodel composition operators. It features a tagging mechanism for storing and keeping organized the metamodels of the involved ADLs. Common semantics are provided by means of a minimalist metamodel containing architectural concepts. The extension of the ADL is performed by composing metamodels via a set of formalized operators. The approach also provides dedicated mechanisms for automatically generating three kind of editors: (i) tree-based, (ii) textual, and (iii) graphical. The reuse of the existing tool of the extended ADL is ensured via automatically generated model transformations (which I call migrators in this dissertation).

• **Evidence on the applicability of the proposed approach in the field.** Parts of the proposed approach have been applied on specific case studies (their description concludes the section dedicated to each part of the approach). Also, the whole
proposed approach has been applied on a real-world case study called BusOnAir\(^7\). The project is about a generic information system for managing real-time information about public transportation systems and making it available as web services. BusOnAir is currently under development as an R&D project at the University of L’Aquila (Italy) and its launch is planned for mid-summer 2012.

- I have developed a series of prototype tools realizing the approaches described so far. Namely, \textsc{Megaf} is the realization of the infrastructure for realizing architecture frameworks, \textsc{Dually} implements the solution for interoperability and synchronization among architecture models, and \textsc{Byadl} is the implementation of the approach for extending and customizing existing ADLs. All the prototypes have been implemented in the context of the Eclipse\(^8\) platform, and can be integrated with other MDE technologies already available in the Eclipse community. Current versions of the tools are available at their corresponding web sites.

\section*{1.6 Structure of this Dissertation}

This dissertation is structured into seven chapters, each of them devoted to describe a specific aspect of my research. Figure 1.3 gives on overview of the structure of this dissertation.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{structure.png}
\caption{Structure of this dissertation}
\end{figure}

\footnotesize
\(^7\)http://www.busonair.eu
\(^8\)http://www.eclipse.org
Basically, after the introduction, this dissertation contains five chapters and a conclusion.

Chapter 2 gives background information on the research areas of this dissertation. More specifically, I will describe concepts about software architecture description and Model-Driven Engineering (MDE).

Chapter 3 presents the proposed solution I envision for modeling the architecture of a software system by reusing, composing and customizing architectural elements. The proposed solution exploits megamodelling techniques with the aim to manage architecture descriptions, to define correspondences among architectural elements, and perform consistency and completeness checks (see Section. Two special mechanisms are employed to (i) assess cross-view consistency and to (ii) extend/customize architectural languages. This chapter is composed of three sections in which I describe in details the sub-project I defined for realizing the overall approach; each section is complemented with a description of a case study in which the corresponding sub-project has been applied.

Chapter 4 presents the details about the concrete realization of the approach described in Chapter 3. It also explains the model-driven technologies I exploit for implementing the approach and how they have been leveraged in order to tackle the research problem object of this dissertation. Furthermore, this chapter discusses how a set of dedicated prototype tools have been implemented for realizing the approach and how they interact with each other in order to provide an infrastructure for defining, reusing, and sharing architectural elements across different projects, organizations, and domains.

Chapter 5 presents a case study in which the proposed approach has been applied. The case study is called BusOnAir and it is about a system for managing real-time information about public transportation systems and making it available as web services. At the end of this chapter, I provide a reflection on the various issues that arose when applying the case study, how they have been handled, and which aspects should be considered and enhanced for future versions of the proposed approach.

Chapter 6 provides an overview about research works that are related to this dissertation. More specifically, this chapter will describe (i) generic MDE approaches focussing on megamodelling, (meta) models composition and weaving, and (ii) SA-specific approaches about architecture frameworks, ADLs interchange and extension/customization.

Chapter 7 closes this dissertation by providing (i) a comprehensive summary of the proposed approach for architecting by means with a focus on reuse, composition and customzation, (ii) final remarks on the results and contributions achieved so far, and (iii) how future research directions will be pursued.
This chapter gives an overview of the basic concepts used in this thesis. Firstly, SA-related concepts like software architecture, architectural concern, ADLs, views, viewpoints, etc. are outlined in Section 2.1. Subsequently, in Section 2.2 I will introduce Model Driven Engineering (MDE) together with its concepts of model, metamodel, model transformation, model weaving, and megamodeling.

2.1 Software Architecture: Concepts and Terminology

Software Architecture is a fundamental element for developing complex systems of any kind, such as service-based systems, embedded real-time systems, web-based systems, etc. The ability to express, analyze, and communicate these architectures is key to that success.\(^1\)

Following a "traditional" definition, a software architecture (SA) consists of a set of components and connectors communicating through interfaces. From a different perspective, an SA consists of a set of architecture design decisions taken to generate the architecture artifact. Indeed, the two definitions are not in contrast, but they are simply orthogonal. In both cases, when an SA is identified, it needs to be described through an Architecture Description Language (ADL). At the core of the architecting activity is architecture description. For this purpose ADLs have been introduced to specify the various aspects of an SA.

Before going into the details about what an SA is, it is important to note that in the past SA was considered the first written-in-stone model of the software system being created. This classical understanding of SA has been refined during the years into a new definition in which SA is considered as a dynamic and incomplete artifact that is always evolving together with the other artifacts along the system development life cycle. This conception of software architecture is nicely summarized by the following quotation from [130]:

\(^1\)http://www.iso-architecture.org/42010/pr-42010-2011-12.html
Chapter 2. Background

Architecture is a compass, not a blueprint.

More specifically, this quotation stresses the fact that SA can always change/evolve with respect to the current solution. So, it should not be seen as a "map" of the to-be system, rather it should be seen as a direction for implementing the system according to its requirements. Further, this quotation highlights the fact that when developing complex systems, it is extremely difficult to produce a complete SA description of those systems; indeed in these cases, SA description can be seen as guidance (i.e., a compass) towards the most acceptable solution, rather than a complete map of the path to undertake towards that solution (i.e., a blueprint).

The recently published ISO/IEC/IEEE 42010:2011, Systems and software engineering Architecture description standard [77], gives a structured and organized overview on the various concepts around architecture description, such as software architecture, system stakeholder, stakeholder’s concerns, architectural views, architecture frameworks, and so on. Also, it specifies best practices for describing architectures to maximize their utility throughout the life cycle. In the following sections I will build on the definitions provided in the ISO/IEC/IEEE 42010 standard to provide a basic overview on the various concepts about software architecture description.

2.1.1 Overview of ISO/IEC/IEEE 42010

ISO/IEC/IEEE 42010:2011, Software and System Engineering — Architecture Description [77], is the internationalized version of IEEE Std 1471, first published in 2000 [76]. The standard addresses architecture description: the practices of recording software, system and enterprise architectures in a consistent form so that they may be understood, documented, analyzed and realized. The standard is method-neutral; designed to be usable by architects employing many different architecting methods.

In active use since its original standardization by IEEE in 2000, ISO/IEC 42010 has been approved as a revised standard by the IEEE-SA Standards Board on 31 October 2011. Whereas the 2000 edition focused on the properties of individual architecture descriptions (ADs), such as what makes an AD complete, consistent, etc., the current revision brings into focus mechanisms for reuse and interoperability of architecting techniques through three mechanisms:

1. Architecture Viewpoints: common ways of expressing and solving a set of known architectural concerns that may be reused across projects;

2. Architecture Description Languages (ADLs): special languages capable of expressing certain system concerns through one or more modeling resources.

3. Architecture Frameworks: coordinated set of viewpoints for use by a particular stakeholder community or domain of application.
2.1 Software Architecture: Concepts and Terminology

Architecture viewpoints, as defined by the standard, codify the practice in architecting of specifying an architecture via multiple views of that architecture where each view is created using some set of conventions, notations and modeling practices. These conventions and associated practices form the viewpoint. The idea dates back to at least the 1970s (Ross’ Structured Analysis) and appears in requirements engineering also [52].

The key idea of an architecture viewpoint is a directed set of modeling resources able to address a particular set of system concerns for a particular audience of system stakeholders. As such, a viewpoint is a form of reusable architectural knowledge (like a pattern or style) for solving a certain kind of architectural description problem with tried-and-tested modeling techniques.

An architecture framework builds on the viewpoint idea as a coordinated set of viewpoints, conventions, principles and practices of architecture description established within a specific domain of application or community of stakeholders.

Similarly, an ADL is a packaging of one or more types of model (usually unified by a common syntax and semantics) enabling certain system concerns to be expressed through one or more types of modeling. Classical ADLs (from the 1990s) include Wright, Rapide, and Darwin [99]. Recent ADLs are ArchiMate, xADL [32] and AADL [48].

The standard specifies requirements on each of these classes of construct that are used to create architecture descriptions with the further goal of promoting understandability, interoperability and overall improvement of the field of architecting by having a common terminology and conceptual basis for these things. To achieve this goal, the standard is built on an ontology of terms and concepts pertaining to architecture description. The concepts include stakeholders, system concerns, views and the viewpoints defining those views, the models of which views are comprised, and the model kinds specifying those models. Finally, all of these elements may be interrelated via correspondences and correspondences rules.

In the following sections I will explain the main concepts contained into the above mentioned ontology: ADs, stakeholders and their concerns are described in Section 2.1.2, architectural views and viewpoints are presented in Section 2.1.3, and architectural correspondences are described in Section 2.1.4; Section 2.1.5 provides basic information about architectural design decisions and rationale. Finally, Section 2.1.6 and Section 2.1.7 provides information about ADLs and architecture frameworks, respectively.
2.1.2 Architecture Description, Stakeholders and Concerns

Figure 2.1 provides a fragment of the ontology defined in ISO/IEC/IEEE 42010 about the context of an Architecture Description (AD). The concept System represents everything that is of interest when architecting. It may contain software entities, hardware components, human beings, processes, materials, and so on. The nature of systems is not defined by the 42010 standard. In the context of this dissertation we can consider a system as the "real-thing" that software architects and developers aim at creating. A system operates within an Environment, that is, everything that may influence the functioning of that system. The environment of a system can contain other systems.

![Figure 2.1: Context of an Architecture description (taken from [77])](image)

As described in ISO/IEC/IEEE 42010, the stakeholders of a system are parties with interests in that system. A stakeholder can be seen as any person that has some kind of relationship with the system of interest; examples of stakeholders encompass architects, developers, designers, testers, end users, etc.

The interests (and expectations) that a stakeholders may have with respect to the system are defined as concerns. Concerns may range from feasibility, known limitations, structure, behaviour, performance, resource utilization, reliability, security, to openness, business goals, strategies, and so on. Stakeholders may define and assign various purposes to a system, they area special kind of concern.

The architecture of a system constitutes what is essential about that system considered in relation to its environment. It is important to note that the ISO/IEC/IEEE 42010 standard makes a distinction between what is an architecture of a system and what is an architecture description: fundamentally, whereas an architecture description is a work product, an

---

2 All the figures taken from the ontology defined in ISO/IEC/IEEE 42010 follow the same conventions for class diagrams defined in [77]
architecture is abstract, consisting of concepts and properties [77]. So, an Architecture Description is a work product used to express the architecture of the system of interest. Depending on the considered stakeholder’s concerns, architecture descriptions may be utilized with different purposes, like: as basis for system design and development, for analysis purposes, for documenting essential aspects of the system, as basis for budgeting and other business activities, and recently also as basis for testing activities [26].

**EXAMPLE OF ARCHITECTURE DESCRIPTION**

Supposing that we want to develop an Adaptive Cruise Control (ACC from now on) for vehicles that automatically adapts its settings with respect to the presence of either other vehicles or road signs; the system of interest is the cruise controller itself, realized as a combination of software modules which coordinate a set of sensors and controllers. The environment of the system is the vehicle on which it is deployed in, the road with the vehicles travelling on it, the road signs along the road, etc.

![Architecture Description of an Adaptive Cruise Controller](image)

Figure 2.2: Architecture Description of an Adaptive Cruise Controller

Figure 2.2 shows an example of architecture description defined as a UML-based diagram opportunely adapted to cope with architectural concepts; the purpose of this model is to highlights the various components making up the system. Possible stakeholders include software developers, the driver, software architects, system engineers; they may have concerns like security, reliability, development costs, maintainability, etc.

**2.1.3 ARCHITECTURE VIEWS AND VIEWPOINTS**

A very important concept that is stressed throughout the ISO/IEC/IEEE 42010 standard is that an architecture description is composed of architecture views and viewpoints. Under
this perspective, an architecture description is not a monolithic artifact, rather it can be seen as a coordinated set of views and viewpoints, each of them focusing on specific aspects of the system under consideration. More specifically, as shown in Figure 2.3, each view is defined in order to address a specific set of corresponding concerns (which in turn are held by a specific set of stakeholders). More specifically, an architecture view is a work product expressing the architecture of a system from the perspective of specific system concerns.

![Diagram](image)

Figure 2.3: Architectural Views and Viewpoints (taken from [77])

It goes without saying that considering the architecture description as made of different views (each addressing a specific concern of the system) helps in managing the inherent complexity of current software systems. For example, software architects may specify a purely structural view to describe the various components of a system, then another view may focus on the development process, another view may focus on more "hardware-related" properties of the system, and other views may focus on security, usability, distribution, and so on.

An architecture view expresses the architecture of the system-of-interest in accordance with an architecture viewpoint. A viewpoint governs its corresponding views, more specifically: the viewpoint establishes the conventions for constructing, interpreting and analysing the view to address concerns framed by that viewpoint. Viewpoint conventions can
include languages, notations, model kinds, design rules, and/or modelling methods, analysis techniques and other operations on views [77]. Summarizing, an architecture viewpoint is a work product establishing the conventions for the construction, interpretation and use of architecture views to frame specific system concerns. An architecture viewpoint may frame one or more concerns and a specific concern can be framed by more than one viewpoint. Examples of viewpoints may include: structural, behavioural, financial, computational, technological viewpoint.

Views and viewpoints are composed of one or more architecture models and model kinds, respectively. Architecture models are specific modeling artifacts that describe the system of interest from a given perspective. Within an architecture description, an architecture model can be a part of more than one architecture view. Architecture models can conform to specific model kinds, that are conventions for a type of modelling. In this context, in the same way as architecture viewpoints govern architecture views, model kinds govern architecture models because they establish conventions for the construction, interpretation and use of architecture models. Examples of model kinds include class diagrams, component-based diagrams, balance sheets, flow diagrams, etc. Architecture models are instances of these model kinds.

**EXAMPLE OF ARCHITECTURE VIEWS AND VIEWPOINTS**

Figure 2.4 builds on the previous example of the Adaptive Cruise Control system and provides two examples of viewpoints that can be used for defining its architecture description. The involved viewpoints are: structural viewpoint for describing the system as composed of software modules, and a behavioural viewpoint for describing the behaviour of each component defined in the structural viewpoint. Those viewpoints are composed of a set of model kinds, each of them establishing the rules and conventions for creating their corresponding architecture models (for the sake of simplicity I did not include those rules in the figure). The involved model kinds are: component diagram, class diagram, and interaction diagram for the structural viewpoint; state machine, Petri net, sequence diagram for what pertains the behavioural viewpoint.

The structural and behavioural views are architecture views and they are governed by the structural and behavioural viewpoints, respectively. The structural view is composed by a UML component diagram governed by the component diagram model kind, whereas the behavioural view is composed by a UML state diagram governed by the state machine model kind. In Figure 2.4 a connection between the state machine diagram and the ACC component exists. The semantics of that connection is that the state machine on the right describes the behaviour of the ACC component; this kind of connections is fundamental for keeping views and viewpoints organized and consistent, they are called architecture correspondences. Correspondences can be governed by correspondence rules defined among viewpoints; correspondence rules are used to enforce and check the various correspondences defined within the architecture description. I will describe both correspondences and correspondence rules in the next section.
Figure 2.4: Example of Architectural Views and Viewpoints

2.1.4 ARCHITECTURAL CORRESPONDENCES

As anticipated above, architectural correspondences can be used to define a relation between any architectural element (AD element in Figure 2.5). An architectural element is any construct that can be part of an architecture description. So, views, viewpoints, stakeholders, concerns, models kinds, etc. can be considered as being an AD element. In this context, it can be seen as the "superclass" of each concept described in the ISO/IEC/IEEE 42010 standard.

From a very abstract point of view, a correspondence can be considered as the equivalent of the concept of "dependency" in UML, since it may be linked to each element within the conceptual model of IOS/IEC/IEEE 42010. Furthermore, as dependencies in UML can be governed by OCL constraints, architectural correspondences can be governed by
correspondence rules. Correspondence rules are used to enforce relations within an architecture description (or between architecture descriptions). In general, the ISO/IEC/IEEE 42010 standard establishes that correspondences and correspondence rules are used to express and enforce architecture relations such as composition, refinement, consistency, traceability, dependency, constraint and obligation. Usually, correspondences and correspondence rules are cross model or cross view since constraints within a model kind can be simply specified within the conventions of that model kind.

EXAMPLE OF CORRESPONDENCE AND CORRESPONDENCE RULE

An example of architectural correspondence has already been anticipated in the previous section (see Figure 2.4, where a correspondence has been defined to put in relation the ACC application component and the state machine diagram describing its behaviour. In this context, a classical example of correspondence rule can be defined as follows:

Each component in a model in a structural view must be related to at least one state machine diagram in the behavioural view.

Clearly, the architecture description described in Figure 2.4 violates this correspondence rule since only ACC application is linked to a state machine diagrams, the other components do not have an associated state machine diagram in the behavioural view. Also, for the sake of understandability in this case I defined the correspondence rule in natural English, the ISO/IEC/IEEE 42010 standard does not pose any constraints on how correspondence rules can be defined. Thus, a correspondence rule may be defined also as an OCL constraint (I will explain how to do it later in this dissertation), a logical constraint, a Java program, and so on.

2.1.5 Architecture Decisions and Rationale

In recent times, software architecture is viewed as a composition of architectural decisions [79] and is receiving much focus as First Class entities, mainly to minimize the vaporization of invaluable architectural knowledge. The ISO/IEC/IEEE 42010 standard defines Architecture Decisions as “position, opinion or judgement reached after consideration of system concerns that pertains to an architecture”[77]. Figure 2.6 shows how architecture decisions related to the other entities within the conceptual model of the 42010 standard.

Architecture decisions may affect any architectural element within the architecture description, and can depend on each other. More specifically, an architecture decision may be in conflict with other decisions, it may be a refinement of another decision, or it may be defined as composition of other sub-decisions, and so on. Also, architectural decisions
may pertain to different stakeholders concerns and, depending on whether a design decision is accepted or not, it may raise also additional concerns to the one already existing in the current architecture description.

The acceptance or rejection of an architecture decision are justified by an architecture rationale. As stated in ISO/IEC/IEEE 42010, an architecture rationale records explanation, justification or reasoning about architecture decisions that have been made. Further, the rationale for a decision can include the basis for a decision, alternatives and trade-offs considered, potential consequences of the decision and citations to sources of additional information.

In the past, architectural decisions seemed to be implicitly hidden in the architecture and this has led to the need for models, methods, tools and classification to explicitly document design decisions. The importance of doing this has been discussed as early as 1986 in the work of Parnas and Clements [119] and in the work of Potts and Burns [125]. Capturing and documenting ADDs has grown from a simple textual representation to a formal process involving standard set of steps. A problem (may be called issue or question alternatively) is identified, several solutions called alternatives are identified, these alternatives are weighed against each other depending on how the impact the resulting artifact and how they address various criteria like security, cost etc., the alternatives are then ranked and the best one among them is chosen and both the choice and rationale are recorded. Several tools are available for capturing ADDs and rationale (Archium [79] and ADDSS [19] for example).

**Example of Architecture Decision and Rationale**

By reconsidering the example provided in Figure 2.2, a possible architecture decision may be the following:
All the data coming from the sensor distributed on the vehicle must be aggregated by a dedicated software module.

This architecture decision may pertain to the performance and reliability concerns, and it affects the whole UML component model of the system. It may also depend on other architecture decisions that can be considered as alternatives. For example, another architecture decision may state that "each physical sensor must be managed by a dedicated software component”; obviously, the last architecture decision is conflicting with our architecture decision, and thus the software architect must choose between them. The rationale that let the software architect to accept the first architecture decision must rely on the fact that having a single component managing all the physical sensors in the vehicle may result in faster communication and synchronization among the sensors.

2.1.6 ARCHITECTURE DESCRIPTION LANGUAGES

According to the ISO/IEC/IEEE 42010 standard, an Architecture Description Language (ADL) is any form of expression for use in architecture descriptions. Figure 2.7 shows how the concept of ADL relates to the other concepts in the ontology defined in the standard.

![Figure 2.7: Architecture Description Language in the 42010 standard (taken from [77])](image)

According to the figure, an ADL provides one or more model kinds as a means to frame some concerns for its audience of stakeholders. An ADL can be narrowly focused, defining a single model kind, or widely focused to provide several model kinds, optionally organized into viewpoints. Often an ADL is supported by automated tools to aid the creation, use and analysis of its models.

ADLs can be classified into three main categories: box-and-line informal drawings, formal architecture description language, and UML and UML-based notations.

Box-and-line have been for a long time the only means for describing SAs. While providing useful documentation, the level of informality limits the usefulness of the architecture description. As remarked in [3]:

---

2.1 Software Architecture: Concepts and Terminology
While these descriptions may provide useful documentation, the current level of informality limits their usefulness. Since it is generally not clear precisely what is meant by such architectural descriptions, it may be impossible to analyze an architecture for consistency or determine non-trivial properties of it. Moreover, there is no way to check that a system implementation is faithful to its architectural design."

A similar conclusion is drawn in [122]:

"Aside from providing clear and precise documentation, the primary purpose of specifications is to provide automated analysis of the documents and to expose various kinds of problems that would otherwise go undetected."

Both seminal papers concluded that what was needed at that time was a more rigorous way for describing SAs.

Since the early 90’s, an outstanding thread of research on formal ADLs has been started, with the definition of tens of different ADLs. As a result, several formal ADLs have been proposed, each characterized by slightly different conceptual architectural elements, different syntax or semantics, focussing on a specific operational domain, or only suitable for different analysis techniques. For example, domain-specific ADLs have been presented to deal with embedded and real-time systems (such as AADL [48], EADL [96], EAST-ADL [29]), control-loop applications (DiaSpec), product line architectures (Koala [146]), and dynamic systems (π-ADL [117]). Analysis-specific ALs have been proposed to deal with availability, reliability, security, resource consumption, data quality and real-time performance analysis (AADL [48]), behavioral analysis (Fractal [15]), and trustworthiness analysis (TADL [113]). However, this remarkable effort has not seen the desired return of industrial practice, for the reasons analyzed in [149, 148, 118, 73, 27]. Formal ADLs have been rarely integrated in the software life-cycle, are only rarely supported by mature and solid tools, scarcely documented, focussing on very specific needs, and leaving no space for extensions enabling the addition of further features.

As a way to overcome some of those limitations, the UML has been indicated as a possible successor of existing ADLs. Many proposals have been presented to use and extend the UML 1.x to model software architectures (e.g., [128, 109, 57, 91, 74, 63, 87, 137]), and then many others for extending the UML2. In fact, while UML 2.x has introduced many new concepts so to make it more suitable for architectural description, still much work has been proposed for extending it according to specific concerns [64, 129, 78, 121]. As a result, a number of UML profiles and extensions have been proposed for modeling different architectural concerns, thus increasing even more the proliferation of architecture description languages.

In general, what can be seen is a tension between more formal approaches to model architectures and UML-based approaches. While formal ADLs are considered (potentially)
valid tools for unambiguously defining SAs, and formally driving the analysis and implementation phases, the implicit complexity of a formal ADL, plus the lack of mature and documented tools, as well as the scarce integration in the production life cycles make formal ADLs only rarely used in industry. On the other side, while UML and its extensions are more easily understood, tool supported, and integrated in the life cycle, the lack of a fully formal semantics [112] and of analysis techniques make them yet another candidate (and not a leader) in architecture descriptions.

EXAMPLES OF ADL

ADLs have been historically classified into two generations [108]. A “first generation” going from 1990 to 2000, had the main purpose to design an ideal ADL [60] whose chief aim was to enable support of components and connectors specification and their overall interconnection [122, 60], as well as composition, abstraction, reusability, configuration, heterogeneity, and analysis [139]. Figure 2.8 shows an example of architecture model defined by using one of the most outstanding ADL of the first generation: Darwin [139].

![Figure 2.8: Example of architectural model in Darwin](image)

![Figure 2.9: Example of architectural model in AADL](image)
Later on, during the “second generation”, going from 2000 up to today, new requirements emerged, and new ADLs have been proposed to deal with more specific features [31, 123], such as configuration management, distribution, and product line modeling support. Structural specifications have been integrated with behavioural ones [102, 56] with the introduction of many formalisms such as pre- and post-conditions, process algebras, statecharts, POSets, CSP, $\pi$-calculus, and others [110]. Figure 2.9 shows an example of architecture model defined by using one a representative second-generation ADL: AADL [48]. Differently from the depicted architecture model in Darwin in which generic components communicate through generic connectors, the architecture model shown in Figure 2.9 provides a more detailed view of the system in terms of components differentiation (e.g., it may be a device like Display, a logical system like HCI\textsubscript{System}, a single process like $P$\textsubscript{HCI} in the figure, etc.), and in terms of their relationships (e.g., here the $P$\textsubscript{HCI} process can communicate to the Display device, it can execute on a specific processor Proc, or it can be connected to other components outside the HCI\textsubscript{System} in order to model their information exchange, etc.). These differences open up a variety of new kinds of analysis, new models visualization means, new way to reason on architecture models.

2.1.7 Architecture Frameworks

The idea of an architecture framework dates back to the 1970s. In enterprise architecture, Zachman popularized the term through his information systems architecture framework [151]. Since then, many frameworks have been proposed, published and used, in a variety of domains and defined with varying degrees of formality. Recent frameworks include RM-ODP, GERAM, TOGAF, and DODAF. Architecting methods are often presented as frameworks, i.e. as a coordinated set of viewpoints to use [94, 75, 131, 25, 42]. The recurring idea among these is that an architecture framework is a prefabricated structure that one can use to organize an architecture description into complementary views [44].

The content model for an architecture framework in accordance with ISO/IEC/IEEE 42010 is illustrated in Figure 2.10. Basically, ISO/IEC/IEEE 42010 extends the original frame of reference for architecture description of IEEE 1471 to define and specify architecture frameworks as first-class entities. Using ISO/IEC/IEEE 42010, an architecture framework is determined by:

1. a set of system concerns arising in the domain of the system under consideration;
2. a set of typical stakeholders holding those system concerns;
3. a set of architecture viewpoints which frame (i.e., cover) those concerns;
4. a set of correspondence rules expressing relations of interest between views, models, and other elements.
In turn, each viewpoint is determined by: (i) the system concern expressible by that viewpoint, (ii) the notations, conventions and model kinds comprising that viewpoint, and (iii) any methods, techniques, and heuristics of use with the viewpoint.

Beyond these minimal requirements linking stakeholders and concerns to a set of viewpoints, and correspondence rules linking views and models together, framework creators are free to add other content to aid users: guidelines, tools, principles, patterns and styles, and so on. For a more detailed discussion of the proposed content model and mechanism for architecture frameworks in ISO/IEC/IEEE 42010, see [44].

**Example of Architecture Framework**

A classical example of architecture framework is the 4+1 Architectural View Model by Philippe Kruchten [94].

It is used for describing the architecture of software-intensive systems and, as shown in Figure 2.11, it is based on the use of 4 concurrent views:

- **Logical view:** the main concern of this view is to define the functioning of the realized system. The model kinds suggested for creating models belonging to this view include, among others, UML sequence and class diagrams. The main stakeholders identified by this view are software architects and designers.

- **Development view:** it is about software development management and the main concern of this view is to present the system from a developer’s point of view. UML component and package diagrams are the model kinds suggested here. The main stakeholders identified by this view are software developers.
Figure 2.11: The 4+1 view model

- **Process view**: the main concern of this view is to consider non-functional properties of the system, like performance and security. This view can be described by using UML activity diagrams as model kinds. The main stakeholders identified by this view are software architects and designers.

- **Physical view**: it concerns the physical distribution of the system being realized, its physical communication infrastructure, and so on. Clearly, UML deployment diagrams are suggested to be used as model kinds. System engineers are the main stakeholders identified by this view.

A fifth special view (the one representing the “+1” in the name of this framework) is devoted to the definition of a set of use cases (or scenarios) with the chief aims to (i) validate the architecture design, (ii) test the various prototypes that may have been defined along the system life cycle, and (iii) glue together all the other views. The model kinds suggested for this view are UML use case and object interaction diagrams. The main stakeholders identified by this view are software architects and designers. Elements of one view are connected to elements in other views, following certain correspondence rules. For the sake of simplicity, I do not repose those correspondence rules in this dissertation; the interested reader can refer to [94] for getting more information about them.

## 2.2 Model-Driven Engineering Concepts

Model-Driven Engineering (MDE [89]) is a promising approach to effectively express domain concepts by abstracting upon selected aspects of the system under consideration and
2.2 Model-driven Engineering Concepts

To assess specific properties of the system early in the life cycle. In MDE, domain-specific modeling languages (DSMLs) are used to describe the application; they are defined using metamodels, which define both the relations between concepts within the domain and their semantics. DSMLs are used to build a model of the system according to the semantics and constraints defined in their metamodel (in this case the model is said to “conform to” the DSML metamodel). It is common in MDE to have a set of transformation engines and generators that, focussing on certain aspects of the models, produce various types of artifacts (e.g., source code, alternative model descriptions, deployment configurations, inputs for analysis tools). As stated in [134], this automated transformation process is often referred to as correct-by-construction, as opposed to conventional software development processes that are tedious and error prone.

In this section I will provide background information on Model Driven Engineering (MDE) together with its related concepts of model, metamodel, model transformation, model weaving, and megamodeling. The structure of this section is as follows: Section 2.2.1 provides the definitions of model and metamodel and how they relate to each other; in this section, UML and its extension mechanisms will be discussed as well. Section 2.2.2 provides a general definition of model transformation and gives an overview of current model transformation approaches by taking into account the classification proposed by Czarnecki et al. in [30]. Section 2.2.3 describes weaving models, that are a special kind of models that I extensively used throughout this dissertation Finally, Section 2.2.4 gives a definition and provides some details on another special kind of model I use in my research: megamodels.

2.2.1 The Metamodeling Stack

It should be clear at this point that models play a key role in model-driven engineering projects. Also, it should be clear that each model must adhere to the rules and constraints defined in its modeling language. The relationship between a model and its modeling language is similar to the relationship that holds between a program and the grammar of the programming language in which it is written [16]. In this respect, the Object Management Group (OMG\(^3\)) has introduced the four-level architecture illustrated in Figure 2.12. The figure shows the various levels of abstraction that exist in MDE and the relationships among them; in the rest of this section I will describe each level of the four-level architecture.

So, starting from the lowest part of the figure, the first level is called M0 and it holds the system being modelled; basically, at this level we are talking about “the real thing”. M0 can be seen as a convenience level, that fundamentally it has been introduced to highlight the represents relationship between the models describing the real system and the system itself.

M1 is about models of the system. Currently there is not an ultimate definition of what a

\(^3\)OMG website: http://www.omg.org
model is. For example, a model has been defined as "a set of statements about a system under study" in [135]; in [10] it is considered as "a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system"; further on, in [111] a model is defined as "a coherent set of formal elements describing something (e.g., a system, bank, phone, or train) built for some purpose that is amenable to a particular form of analysis". I believe that the following is one of the most concrete definitions of what a model is: a model is a reduced representation of some system that highlights the properties of interest from a given viewpoint [138] since it builds on two fundamental points in MDE: models should be a reduced representation of the system, and models should have a focus in order to be useful to the engineer. In general, a model may represent system requirements, the design specification or even the program code (this is a peculiarity of the Model Driven community). A given system may have $n$ different models, each model representing a specific aspect of the system.

The M2 level contains metamodels; every model must conform to a specific metamodel that defines its semantics and the elements that may be specified into the model. A more detailed definition is that a metamodel defines the first-class entities, relationships and constraints of a modeling language.

Figure 2.12 presents a third modeling level (M3) containing metamodels for metamodels, they are called Meta-metamodels. Since the conforms to relation does not have any constraints on the modeling elements it links, the metamodeling stack may be ideally infinite; however, in order to give a more concrete setting to the stack, and in order to make the...
implementation of modeling tools more easy to understand and maintain, it has been decided to make the M3 modeling level self-referencing. That is, the there is no other level above it in the metamodeling stack since metametamodels conform only to themselves.

2.2.2 Model Transformations

If on one hand modeling is fundamental when abstracting the details of a real system and reducing the complexity of software systems during their development, design and maintenance; on the other hand model transformations are fundamental for elevating models from documentation elements to first-class artifacts of the development process.

In the following I provide a basic definition of model transformation. This definition is taken from the ATL\(^4\) (a specific language for model transformations) user guide \([67]\) because it is at the same time concise and easy to understand.

A model transformation aims to provide a means to specify the way to produce target models from a number of source models. For this purpose, it should enable developers to define the way source model elements must be matched and navigated in order to initialize the target model elements. Formally, a simple model transformation has to define the way for generating a model \(M_b\), conforming to a metamodel \(MM_b\), from a model \(M_a\) conforming to a metamodel \(MM_a\). The model transformation itself therefore has to be defined as a model. This transformation model has to conform to a transformation metamodel that defines the model transformation semantics. As other metamodels, the transformation metamodel has, in turn, to conform to the considered metametamodel.

Figure 2.13 summarizes the a generic model transformation process. A model \(M_a\), conforming to a metamodel \(MM_a\), is transformed into a model \(M_b\) that conforms to a metamodel \(MM_b\). The transformation is defined by the model transformation model \(M_t\) which itself conforms to a model transformation metamodel \(MM_t\). This last metamodel, along with the \(MM_a\) and \(MM_b\) metamodels, has to conform to a metametamodel \(MMM\) (such as MOF, see Section 2.2.1).

An important distinction of model transformations is that a model can be transformed either horizontally or vertically. The former means that the source model is transformed into a model on the same level of abstraction (horizontal transformation), the latter means that the source model is transformed into a model on another level of abstraction (vertical transformation).

Also, it is important to note that currently model transformations can be used within a variety of technical spaces, including modelware (when transforming between models),

\(^4\)ATL project website: http://eclipse.org/atl/
grammarware (when at least one of the involved models is represented as a textual specification that conforms to a grammar), or XML-ware ((when at least one of the involved models is represented as an XML file that conforms to an XML schema). In this work, if not differently specified, I assume that the technical space of the models involved in some transformation is that of modelware.

In the following I will briefly present the state of the art about model-to-model transformation approaches. In [30] Czarnecki and Helsen propose a feature-based framework for the classification of current available model transformation approaches. Because of the introductory nature of this chapter I do not describe their classification framework, I will mainly give an overview of the actual approaches that they surveyed.

In [30] the first level of classification is composed of model-to-text, model-to-model and text-to-model approaches. Transformations of the first type create their target as an instance of the target metamodel. The target of second-type transformations (e.g. code or specification generators) is just strings and conversely the target of third-type transformations (e.g., reverse engineering or parsing tools) is a model, while their source is strings. For the sake of generality, in this section I deal only with model-to-model transformations; this is due to the fact that other types of approach can be viewed as a special case of model-to-model transformations (e.g., we can assume that a metamodel that "superstands" the text in input/output may be provided).

Czarnecki and Helsen surveyed seven model-to-model transformation approaches:

1. **Direct manipulation** (JMI): this is the most low-level approach, offering an internal model representation and some APIs to manipulate models.

2. **Structure Driven** (OptimalJ): this is a two-phases approach. In the first phase the hierarchical structure of the target model is created; in the second phase the internal features of the target elements are set. Users have only to provide the transformation
2.2 Model-driven Engineering Concepts

rules.

3. **Operational** (Kermeta, MTL): this is similar to the Direct approach but the metamodeling formalism is extended with facilities for expressing computations.

4. **Template-based**: this approach is based on the concept of model templates (models annotated with metacode that computes the varying parts of the instance of the template). The metalanguage is usually expressed via OCL expressions.

5. **Relational** (QVT Relations, AMW): the basic idea of this approach is to specify the relations between source and target element types using constraints (relations, mappings, logic formulas). This approach is very attractive because of its strong theoretical basis, but in most cases performance of transformation strongly depends on the kinds of constraints that need to be solved.

6. **Graph Transformation** (VIATRA, Fujaba): this approach is inspired by theoretical work in graph transformations. Graph transformations are declarative and also seem intuitive; however, this approach suffers the possibility of lack of confluence and termination. Recent Research works are being developed in order to tackle this problem. Moreover another weakness of this approach is that current graph-transformation theories do not consider ordered graphs. Therefore this approach do not apply to method bodies or functions, in these cases the order of the executed statements is ignored.

7. **Hybrid** (QVT, ATL): this kind of approach consists of the merging of two or more previously defined categories, depending on the application.

8. **XSLT-based** (XSLT): models are often serialized as XML files, this makes XSLT to be a good candidate for implementing model transformations. This is true, but this approach has scalability limitations caused by the verbosity and poor readability of XML and XSLT.

Czarnecki and Helsen conclude the survey declaring that Hybrid is likely a comprehensive approach because the user may adapt the transformations according to the application and the models to transform. This is one of the main motivations that drove me to choose ATL for implementing the parts of my approach in which models must be transformed.

2.2.3 MODEL WEAVING

Weaving models, as their name suggest, are special kinds of models that allow engineers to link together other models. So, in general a weaving model contains a set of links between elements of a model and elements of another model. In this respect, the linked models are called "woven models" and, by recalling the four-levels architecture described in the previous section, it goes without saying that woven models can be either terminal
model weaving is a generic operation that establishes fine-grained correspondences between model elements [46], each correspondence defines associations between elements from different models.

Figure 2.14: Weaving model

Figure 2.14 shows a weaving model $WM$ linking the concepts of two metamodels ($MM_a$ and $MM_b$, respectively). Each weaving link within $WM$ can potentially link every kind of element within the metamodels: metaclasses, attributes, references, data types, etc. This makes weaving models a very flexible means to represent the various relationships that may exist between the woven models.

As any other model-driven artifact, also weaving models must conform to a metamodel ($WMM$ in the figure); in this case it is called the “weaving metamodel”. A weaving metamodel specifies the types of links that can be established between woven models. Typical examples of link types defined in a weaving metamodel include:

- trace link,
- dependency link,
- equals,
- etc.

The model weaving operation offers a number of advantages. First of all, all the information, relationships and correspondences between the considered models, can be described by specialized weaving models avoiding to have large metamodels for capturing all the aspects of a system [35]. Furthermore, different metamodels focusing on their own domain can be individually maintained, and at the same time interconnected into a "attice of metamodels"[16]. In other words, each metamodel could represent a domain-specific language dealing with a particular view of a system, while weaving links allows developers to describe the aspects both separately and in combination.
In the context of this work, weaving models are fundamental in numerous situations; indeed I use them to:

- represent the various correspondences that may exist between architecture models pertaining to different architectural views (see Section 3.1);
- navigate those correspondences in order to perform correctness and completeness checks in an architecture framework (see Section 3.1);
- define pro-active inter-view correspondences that semi-automatically keep views in a consistent state (see Section 3.2);
- keep tracing information while transforming architecture models while passing from an architecture view to another (see Section 3.2);
- extend ADLs metamodels by means of semi-automatic model transformation techniques (see Section 3.3);

2.2.4 MEGAMODELING

Megamodeling [12] has been proposed with the aim of supporting modeling in the large, i.e. dealing with models, metamodels, and their properties and relations. Intuitively, a megamodel is a model of which at least some elements represent and/or refer to models or metamodels. While a metamodel specifies properties and rules governing models construction, a megamodel specifies properties and rules governing Model Driven Engineering (MDE) artifacts construction, and among them, models and metamodels. Megamodeling operations support the management of large libraries of artifacts that, as said before, could also be models and metamodels.

Megamodeling offers the possibility to specify relationships between models (and metamodels) and to navigate among them. This is fundamental in MEGAF since architectural views generally have important relations defined among them. Furthermore, in order to have a homogeneous infrastructure, I assume that all the used heterogeneous artifacts are models and that each model conforms to its metamodel. This enables the management of complex artifacts since their complexity is defined and encoded in the metamodel, thus enabling programmatic management of (even complex) models. This assumption follows a basic tenet of the ISO/IEC/IEEE 42010 standard: each view conforms to a viewpoint, and each architecture model conforms to a model kind. The assumption also seems reasonable in light of the recent Doc2Model (Document to Model) Eclipse project for parsing structured documents to produce EMF models.

Figure 2.15 shows an excerpt of the megamodeling conceptual framework, as presented in [4]. In the following of this dissertation I refer to a metamodel of a megamodel as metamegamodel. The main types of models are:

\[\text{http://eclipse.org/proposals/doc2model}\]
\[\text{http://www.eclipse.org/modeling/emf}\]
Figure 2.15: Basic Megamodelling Conceptual Framework: modeling artifacts

- **Terminal Models**, which represent the real system and conform to some metamodels;
- **MetaModels**, which define domain-specific concepts and conform to metamodels;
- **MetaMetamodels**, which provide generic concepts for metamodels specification and conform to themselves.

**MetaModels** and **MetaMetamodels** are called **Reference Models** since they define the concepts that can be instantiated in other models; so they can be considered as a reference for all the models conforming to them. Figure 2.15 shows different kinds of terminal models:

- **Transformation Models**, which are used to define transformations between models;
- **Weaving Models**, which are used to define relationships among models;
- **MegaModels**, which are used to support the megamodelling process.

It is important to note that **Terminal Model** is not an abstract metaclass, and then it can be instantiated as well in order to represent the various models of the system under consideration; in this case the terminal model must conform to and only to a single metamodel.

As introduces before, megamodelling allows also to specify (typed) relationships between the modeling artifacts contained into a megamodel. Figure 2.16 shows another excerpt of the megamodelling conceptual framework representing the artifacts relationships. They represent the generic relationships that may exist between the various entities within the megamodel. An **Entity** is the root metaclass of the generic metamegamodel, and represents any possible concept that may exist within a megamodel. A relationship may be either bidirectional (**Relationship** in Figure 2.16) or directed (**DirectedRelationship** in Figure 2.16), i.e., if it distinguishes or not between the elements it links.

In the literature, the megamodelling conceptual framework described above has been extended in many ways and for different purposed. For example it has been extended for
2.2 Model-driven Engineering Concepts

Figure 2.16: Basic Megamodelling Conceptual Framework: relationships

supporting a process for doing model-driven performance analysis [54], for keeping tracking information after the execution of model transformations, or to reverse engineer a given platform from available metadata [103]. In particular, a recent extension proposes the use of megamodels in combination with weaving models [86] for coordinating sets of models. In that work, on one hand a megamodel represents all models involved in a given context as well as various relationships between them. On the other hand, weaving models are used to represent different kinds of fine-grained relationships between the various elements contained in the models.

Figure 2.17: An example of megamodel with navigation and traceability links

Figure 2.17 is taken from [86] and represents a megamodel in which navigation and traceability links are used to coordinate a set of models. The megamodel in figure contains three different terminal models (Ma, Mb, and Mc), each conforming to a different metamodel (MMa, MMb, MMc in the figure). Model-level relations (MW Relation in the
figure) hold between $Ma-Mb$ and $Mb-Mc$; these relations can be used to navigate between the various models and to specify that there is some kind of "tracing" relationship between them. Then, those relationships are refined into two weaving models (weaving models $a-b$ and $b-c$ in the figure) that contain the needed relations that link specific model elements within the models.

The navigability and traceability extension had a strong impact on my work since it allows software architects to define relations between architectural models at two levels of abstraction: model-level and model element level, thus promoting a definition of more accurate correspondences between the architectural elements involved in MEGAF.

As a concrete implementation of megamodelling, I refer to the AMMA platform\footnote{http://www.sciences.univ-nantes.fr/lina/atl/AMMAROOT}, as presented in [11]. More specifically, a dedicated component of the AMMA platform is in charge of managing megamodels, it is called AM3 [4]. In AM3, a megamodel records all available resources and acts as an MDE repository. From a practical point of view, AM3 manages megamodel elements (for example model transformations, tools, UML models, metamodels) and provides a user interface to manipulate them. This user interface is generic since it does not depend either on the models, metamodels or other artifacts within the megamodel. AM3 is extensible, and thus it allows developers to extend the base metamegamodel by providing new concepts in a separate metamegamodel.

Under this perspective, software architecture concepts can be defined as an extension of the base metamegamodel; in so doing, it is possible to navigate SA-related artifacts and establish (typed) correspondences among them by reusing the AM3 extensible platform. In Section 4.3, I will describe a new extension that is specifically focused on the management of the various entities of the conceptual model of the ISO/IEC/IEEE 42010 standard.
In the previous chapters I discussed the main motivations at the basis of the proposed solution and I gave an overview about software architecture modeling and model-driven engineering techniques. In this chapter I will give the details about the proposed solution I envision for modeling the architecture of a software system by reusing, composing and customizing architectural elements like architectural languages, views, viewpoints, and so on.

As anticipated in Section 1.4, the proposed solution is called MEGAF and leverages megamodeling techniques with the aim to (i) create, store, and manage viewpoints, views, and concerns, (ii) define correspondences among architectural elements, and (iii) perform consistency and completeness checks. Figure 3.1 shows the conceptual overview of the MEGAF approach and highlights its relationship with respect to the other approaches presented in this dissertation. More specifically, in the context of MEGAF, two special mechanisms are employed to assess cross-view consistency and to extend/customize architectural languages; they are called DUALLY and BYADL, respectively.

It is important to note that MEGAF provides to both software architects and researchers in the SA field a generic means to precisely define SA concepts and their relationships. Thus, it can be considered as a starting point for the definition of new approaches in the field of software architecture in which a precise definition of the various relationships between SA concepts is needed. Indeed, those relationships can be represented in MEGAF by means of a dedicated model (which can be seen as a "map" of all the SA concepts within MEGAF), and thus they can be navigated, populated, manipulated, or analysed by other engines running on MEGAF. This is exactly what has been done with DUALLY and BYADL: they make use of the MEGAF infrastructure for representing SA concepts and
then they navigate and manipulate elements within that infrastructure for tackling the architecture models consistency and ADLs adaptation problems, respectively. MEGAF has been designed as extensible so that future approaches in the SA research field can benefit from its provided infrastructure, in a way similar to what has been done in DUALLY and BYADL.

Figure 3.2 shows one of the possible usage scenarios of the proposed approach and how its single parts (i.e., MEGAF, DUALLY and BYADL) can support different phases of the architecting activity. Fundamentally, we can consider the architecting activity as composed of two main phases:

1. **Architecture Framework definition.** In this phase architects use MEGAF to establish the architecture framework that better fits with the system being architected or with its domain of interest. The architecture framework is defined in order to precisely focus on the specific peculiarities and technological aspects of the system being considered. The various steps of this part of the usage scenario are self-explaining, so I will not describe them in this part of dissertation (please refer to Section 2.1.1 for a description of each of them). However, it is important to detail the two steps in which BYADL and DUALLY can be used. BYADL can be used (if needed) for extending or customizing the selected ADLs in order to better fit with some aspects of the system being modelled (see Section 3.3.1 for more details on how to do this); whereas, DUALLY can be used to define proactive correspondence rules that can be used later during the architecture description activity for checking and enforcing architecture models consistency among views (see Section 3.2.1 for more details on this part of the proposed approach).

2. **Architecture Description definition.** In this phase MEGAF allows architects to use the created architecture framework to define an architecture description of the sys-
3.1 MEGAF: an infrastructure for realizing architecture frameworks

As discussed in Section 2.1.1, the ISO/IEC/IEEE 42010 standard [77] defines a conceptual model of architecture description. In this research I exploit megamodeling techniques to put in a concrete form the concepts and guidelines expressed in the ISO/IEC/IEEE 42010 standard. The result is MEGAF [72], that is an infrastructure for realizing architecture frameworks which can be used to create architecture descriptions. By referring to Figure 3.3:

- MEGAF is an extensible repository of viewpoints, views, model kinds, architecture models, system concerns, and stakeholders.

- Correspondences and correspondence rules between arbitrary elements, including stakeholders, system concerns, models, etc. can be created in MEGAF. More precisely, they enable the architect to express and enforce relations both between various elements inside an architecture description and across architecture descriptions (such as for product lines or systems of systems). The role and variety of uses of correspondences has been studied extensively [14].

- MEGAF provides functionalities that allow software architects to create their own framework by properly selecting among artifacts previously defined and resident inside MEGAF.
Once the framework has been defined, it can be used to realize the architecture description of the system-of-interest. In other words, MEGAF allows the architecture description to be created compliant to the architecture framework, such as the realized architecture models conform to suitable model kinds contained in the architecture framework; moreover views are governed by viewpoints defined in the architecture framework and address some previously-specified system concerns.

Software architects can benefit from MEGAF in three possible ways:

1. Software architects (and their organizations) that use a predefined architecture framework. They do not create new viewpoints, but use an existing framework “out of the box”. However, they may need to customize the presentation of architecture descriptions for various stakeholder audiences or integrate that framework with existing project tools or artifacts.

2. Software architects may create new viewpoints or model kinds to address particular system concerns (e.g., fault tolerance) not addressed by a current framework. They would then integrate those by linking them with other existing representations.

3. Academic researchers on software architecture develop new analysis techniques or representations on top of existing base derived from the previous two cases.

Also, as discussed in Section 2.1.1, the ISO/IEC/IEEE 42010 standard [77] introduces the concepts of correspondences and correspondence rules to enable the architect to express
and enforce relations between various elements (called *AD Elements*) in an architecture description. The typical case is to relate items within one view or model to items in another but this is generalized to correspondences between arbitrary elements, including stakeholders, system concerns, models, etc. to be expressed as well. The correspondence mechanism is also useful for expressing relationships *across* architecture descriptions (such as for product lines or systems of systems). The role and variety of uses of correspondences has been studied extensively [14].

### 3.1.1 Conceptual Overview of MEGAF

The initial step for building MEGAF is the creation of a generic metamodel for software architecture megamodels; this metamodel is called GMM4SA, that stands for Global Model Management for Software Architectures. Models conforming to GMM4SA have other models as first class entities, such as architectural models, their metamodels, external models for representing correspondences, and so on; views, viewpoints, stakeholders, and stakeholder’s concerns are also part of the megamodel. Developing a metamegamodel for software architectures allows to:

- represent architectural elements as model instances,
- define relationships and correspondences among architecture elements in a natural and straightforward way,
- provide a suitable engine for automating activities related to software architecture elements (see Section 4.3 for more details about this aspect of MEGAF).

Relationships expressed in the ISO/IEC/IEEE 42010 standard have been encoded into GMM4SA, so that each megamodel conforming to it must satisfy those relationships in order to be valid. Additional relationships and rules, specific to an architecture framework or organization are defined by means of weaving models and OCL\(^1\) specifications; these implement correspondences and correspondence rules, respectively. Therefore, each architecture description (AD) element (e.g. view, viewpoint, stakeholder, concern, etc.) can be explicitly represented via a model or a model element depending on its granularity (e.g. the *SystemConcern* becomes a metaclass that can be instantiated as many times as needed to model different concerns, whereas *ArchitectureModel* becomes a metaclass that references an external resource, so its instances are externally defined models).

Figure 3.4 presents an excerpt of GMM4SA. GMM4SA is defined as an extension of an already existing metamegamodels developed in the context of AM3 (its elements appear colored). The metamegamodel I use as starting point for developing GMM4SA has been already discussed in Section 2.2.4. In GMM4SA five metaclasses extend concepts belonging to the base metamegamodel:

---

Figure 3.4: Extract of GMM4SA, Megametamodel for Software Architectures
3.1 **MEGA4F: an infrastructure for realizing architecture frameworks**

(i) **ADEElement** is an abstract metaclass representing generic architecture description elements in the sense of the ISO/IEC/IEEE 42010 standard. All metaclasses within the GMM4SA metamodel extend ADElement (for the sake of readability this relationship is omitted from the figure). **ADEElement** inherits from **IdentifiedElement** and **Entity** of AM3, so that it has a unique identifier, may be linked to other identified elements and may be navigated via the AM3 entities navigator.

(ii) **ArchitectureDescriptionLanguage** inherits from **Metamodel** in AM3 since an architecture description language can be represented as a metamodel to which an **ArchitectureModel** may conform. By doing this, I can reuse standard model-driven mechanisms to check whether an architecture model conforms to its corresponding architecture description language.

(iii) **ArchitectureModel** inherits from **TerminalModel** in AM3. This allows to store architecture description languages and architecture models as external models, thus promoting their reuse.

(iv) **CorrespondenceRule** inherits from **LocatedElement** of AM3 since in MEGA4F a correspondence rule is specified with OCL rules defined in an external (and so to be located) file.

(v) Finally, **Correspondence** inherits from the **ModelWeaving** metaclass of the basic metamegamodel since a correspondence is a specific kind of weaving relationship; it refers to a weaving model linking two or more architecture description elements.

In the following I present the metaclasses of GMM4SA that do not extend any external metamodel. **ArchitectureFramework** and **ArchitectureDescription** are the main elements of GMM4SA. An **ArchitectureFramework** has an associated set of **ArchitectureViewpoints** that define a set of **ModelKinds**, a **Convention** for each **ModelKind**, a set of **SystemConcerns**, a set of **Stakeholders**, and finally a set of **CorrespondenceRules**.

A **ModelKind** is a special kind of **ADEElement** that can be associated to either an **ArchitectureFramework** or an **ArchitectureDescriptionLanguage**.

An **ArchitectureDescription** has an associated set of **ArchitectureModels** each conforming to a **ModelKind**, a set of **ArchitectureViews** organizing those **ArchitectureModels** and addressing some system concerns held by stakeholders, thereby expressing the architecture of the system-of-interest each in accordance with a viewpoint, **Correspondences** and **CorrespondenceRules**. Moreover, an **ArchitectureDescription** offers a set of **ArchitectureRationales** to justify **ArchitectureDecisions**. The GMM4SA metamodel is complemented with a set of OCL constraints that ensure properties that are not purely structural; in the following I elaborate on this part of the metamegamodel.

It should be noted that there are three metaclasses whose instances are external models, they are:
• **ArchitectureModel** is a specific kind of terminal model.

• **ArchitectureDescriptionLanguage** extends the concept of metamodel ².

• **Correspondence** is a specific kind of weaving relationship; it refers to a weaving model linking two or more architecture description elements.

• **CorrespondenceRule** is considered as a terminal model. This is due to the fact that a correspondence rule may be defined as an OCL specification ruling on architectural elements in the megamodel.

As described before, GMM4SA offers two mechanisms to define correspondences among AD elements: the first is **Correspondence** that allows software architects to define weaving relationships between two metamodels; the second is **CorrespondenceRules** expressed in terms of OCL rules. These rules can be defined between metamodels or models. OCL rules are extremely important to enable consistency and completeness checks.

**MEGA** supports different kinds of checks:

- Conformance of an architecture framework with the ISO/IEC/IEEE 42010 standard: GMM4SA encodes the rules of the standard and thus the conformity of a megamodel to the standard is partially ensured via the conformity of the megamodel to GMM4SA; moreover, additional OCL rules must be added since the metamodel only specifies structural aspects. For instance, a rule may be added to ensure that for each model kind, suitable conventions have been defined (see Listing 3.1).

```
context ModelKind
inv is_convention_defined :
  Convention.allInstances() -> exists(e | e.modelKind = self)
```

- The standard also specifies other desired checks that have been encoded and automated in MEGA. For instance, even though an architecture description could conform to zero, one or more architecture frameworks, it is useful to have automation to check the compliance of an architecture description with a particular architecture framework. The standard defines requirements (cf. [77], p.19) to ensure that each stakeholder, concern, and viewpoint identified in the architecture framework has been considered in the architecture description, and moreover that each correspondence rule identified in the architecture framework holds in the architecture description.

- Additional rules can be defined as part of a specific architecture framework. For instance, one would like to ensure that for each model kind defined in a specific viewpoint, a model conforming to the model kind is defined in the associated view (see Listing 3.2).

---

²By doing this each architecture model can conform to a specific architecture description language, and thus I can exploit MDE techniques to check whether an architecture model conforms with its architecture description language.
3.1 **MEGAf: an infrastructure for realizing architecture frameworks**

Listing 3.2: Example of framework-specific OCL constraint.

```
context ArchitectureViewpoint
inv are_adls_instantiated :
  self.adls ->forall (i | ArchitectureModel.allInstances().->exists (e | e.conformsTo=i))
```

Finally, software architects can define correspondence rules, using OCL rules, on the architecture description as required by the system-of-interest. For instance, rules can be defined to ensure the consistency among different views and among models defined inside a view.

3.1.2 **The SBSCS Example**

In this section I describe an example application to show how an architecture description and an architecture framework may be modelled using our approach. The running example is a Subscription-based Sensor Collection Service (SBSCS) architecture description; it is a simple architecture description (the “Hello World” of ADs!)\(^3\). In particular, I model an ad-hoc architecture framework for the SBSCS system and then I provide an architecture description of SBSCS by applying the defined framework.

Figure 3.5: Megamodel for the running example SBSCS

Figure 3.5 shows an overview of a megamodel defined for the running example SBSCS that conforms to GMM4SA. The megamodel is conceptually divided into two parts, the architecture framework and the architecture description. The architecture framework contains three viewpoints, namely the financial, the operational and the system viewpoints. Each viewpoint governs a view that is referenced in the architecture description. Associated to each viewpoint there is an architecture description language, as shown in figure.

\(^3\)http://www.iso-architecture.org/ieee-1471/docs/SBSCS-AD-v02.pdf
Architecture description languages are described by a set of common information like overview, description and references to external documents.

Moreover, each architecture description language is completed by its underlying metamodel. Architecture models conforming to these metamodels are part of the architecture description. As can be seen in the figure, the SCS Dataflow architecture model is realized with a graphical editor built for the SID model kind. The Collection TLD architecture model is realized by using the default tree-like editor. Finally, the SCP profit statement architecture model is realized by means of an excel sheet, since MEGA has been instrumented with ATL transformations able to import/export excel files.

A correspondence, i.e. a weaving model, is defined to relate nodes in the dataflow diagram to elements in the timeline diagram. The ATL query in Listing 3.3 encodes the OCL constraint specifying the NodeCheck correspondence rule: “each node in a data flow diagram must appear at least once in the timeline diagram with its corresponding response time for that node”.

Listing 3.3: NodeCheck correspondence rule as an ATL query.

```java
query NodeCheck = 
  if(SID! Performer . allInstancesFrom( 'IN' ) . exists( e | not e . timelineNodeExists() ))
    then 'NodeCheck is violated, violating nodes:' +
    SID! Performer . allInstancesFrom( 'IN' ) . select( e | not e . timelineNodeExists() ).
toString()
  else 'NodeCheck is not violated.'
endif.println();
```

More specifically, in line 2 the NodeCheck query considers all the performer nodes in the dataflow diagram (i.e. SID in this context) and checks if there is some node that do not have a corresponding entry in the timeline diagram. If it is so (lines 3-4), then all the nodes violating the rule are printed to the console, otherwise an informative message is simply printed to the console (lines 5-6). The timelineNodeExists helper returns true if the weaving model between the dataflow and the timeline diagrams (i.e. WEAVING in line 9) contains a link between the current performer node (line 9) and some entry in the timeline diagram (line 10).

Both the weaving model and the OCL constraint are needed because the weaving model serves to store the correspondences among the dataflow and the timeline diagram elements, whereas the OCL constraint is used to automatically check if the correspondence rule holds between the two models; this is done by systematically navigate weaving model linking them.

In this simple example each viewpoint frames a system concern and each system concern is addressed by a view. Finally, the architecture description identifies only three of the eight stakeholders associated to the framework; this is because the other stakeholders do not have any concern related to the SBSCS architecture being described.
3.2 Dually: A Solution for Providing Architecture Description Languages Interoperability

As highlighted in the official ISO/IEC/IEEE 42010 website[^4], the use of multiple views for architecting can get very complicated. Their consistency is a potential problem whenever multiple models and views are used. Sometimes, consistency rules or procedures are defined as a part of viewpoints. In other cases, organizations will have practices they use to check and enforce consistency. The solution provided by the ISO/IEC/IEEE 42010 standard consists in using correspondences and correspondence rules to define the various relationships that may exist within an architecture description.

Under this perspective, in this dissertation I propose **Dually** as an approach in which special kinds of correspondence rules can keep in a consistent state architecture models belonging to different views. Those special correspondence rules are defined between ADLs and have the additional feature of being proactive; that is, when an inconsistency is detected between different architectural models, the **Dually** interoperability engine actively transforms the models in order to restore their consistency. Consistently supporting ADLs interoperability and change propagation is intrinsically complex. Furthermore, the lack of automation does not allow the easy addition of new description languages, and does not guarantee change propagation to multiple models in a finite number of steps. In general, changes occurring in an architecture model have a strong impact on all the other related architecture models (each of them possibly conforming to different architecture description languages). In order to keep models in a consistent state, changes need to be propagated from the updated model to all the others. When dealing with multiple architecture description languages, propagating changes may be a complex task; such a task is inevitable and requires to be managed by a dedicated approach.

**Dually** [106] enables the transformation of a model conforming to a specific architecture description language into corresponding models conforming to other architecture description languages. In **Dually** the interoperability among various architecture description languages is ensured via model transformation techniques.

Instead of creating a point-to-point relationship among all languages, **Dually** defines the transformations among architecture description languages by passing through $A_0$, which is a core set of architectural concepts defined as generally as possible (to potentially represent and support any kind of architectural representation) and extensible (in order to add domain specificities). In other words, $A_0$ acts as a bridge among the different architectural languages to be related together. The star architecture of **Dually** enables an agile and easy integration of architecture description languages.

The **Dually** transformation system is made of a series of model-to-model transformations that enable information migration among architecture models. These model-to-

[^4]: http://www.iso-architecture.org/42010/faq.html
model transformations are constructed automatically by executing higher-order transformations (i.e., transformations taking other transformations as input or producing other transformations as output).

While DUALLY transforms a model into any other by passing first through an A0 model, model changes are propagated accordingly first to the A0 model and successively forwarded to any other architectural model (it has to be noted that the obtained result is independent from the order followed in the forwarding). Under the assumption that concurrent modifications to different models cannot apply, the DUALLY architecture ensures the convergence of the change propagation process, that is, it ensures by construction that a modification of a model within the network is propagated to all the other models in a finite number of steps (future work in order to manage concurrent modifications is discussed in Section 7.1).

![Architecture models interoperability: generic process (a) and through A0 (b)](image)

In order to give a clear idea on the benefits of using the A0 pivot language, Figure 3.6 shows two possible change propagation processes: the change propagation in a generic process and the change propagation as regulated by A0. By referring to Figure 3.6.a (i.e., the generic changes propagation process) once model D is modified to D’, the changes must be propagated to A, B, and C. However, as soon as B is modified due to the modification in D, the changes in B must be propagated to A, C, and even D. The same holds for changes caused in A and C. This simple example provides the intuition that the change propagation output strongly depends on the order in which models are updated after a change, and that the propagation process could easily diverge. By referring to Figure 3.6.b, the changes propagation regulated by A0 fixes these problems. Once a model D has been modified to D’, changes are propagated to A0, and successively from A0 to A, B, and C, following any order. The changes propagation output is independent from the followed order. This is possible because of the star topology in which notations are arranged: once D’s changes are propagated to A0, the updated A0 model is considered as "frozen" and the propagation of changes to each peripheral model (A, B, and C in the figure) is performed independently. Under the assumption that no concurrent changes are allowed, after applying those changes propagation, new modifications can be performed.

In next sections I will firstly describe how DUALLY allows architects to (semi)-automatically generate the needed model transformations between architecture description languages, and how those transformations can be included and used into the MEGAF meg-
3.2 DUALLY: a solution for providing architecture description languages interoperability

amodel (Section 3.2.1), then in Section 3.2.2 I will describe the concepts that are part of the $A_0$ pivot language.

3.2.1 THE DUALLY MODEL TRANSFORMATIONS ENGINE

Figure 3.7 shows a high level conceptual view of how ADLs are arranged in DUALLY. This figure also puts in evidence the main purpose of DUALLY, that of allowing different ADLs to interoperate (together with their related tools). Deeper down to the conceptual details, Figure 3.8 presents an instance of two branches from Figure 3.7’s more general view.

![Figure 3.7: DUALLY High Level Conceptual View](image)

To enable a precise and meaningful information exchange between the involved ADLs, precise and powerful transformations must be devised. In DUALLY, the interconnections depicted in Figure 3.8 exist at two different levels of abstraction: they can be considered as either model-to-model transformations or as metamodel-to-metamodel links.

The star architecture of DUALLY permits an agile and easy integration of more and more technologies as the need rises. As previously stated, the transformation system of DUALLY is made of a series of low level model-to-model transformations that enable information migration among architecture models. These model-to-model transformations are constructed automatically executing a set of higher-order transformations (i.e., transformations taking other transformations as input or producing other transformations as output). Such transformations utilize information provided by the metamodel-to-metamodel links that link the various DUALLYzed ADLs.

The presence of weaving models represents a strong benefit for the DUALLY approach. Indeed, weaving models constitute a layer that makes the whole model transformation engine totally transparent to software architects. Software architects just have to create a weaving model to define the correspondences between an ADL metamodel and $A_0$; DUALLY will automatically transform the weaving model into the specific model transformation code implementing the transformation. Fundamentally, the process to create and use the weaving model in DUALLY is the following:
1. creating (or importing, see Section 4.2) the ADL metamodel into the DUALLY platform;

2. graphically developing the weaving model between the ADL metamodel and $A_0$ through the DUALLY weaving models editor;

3. applying the DUALLY transformation engine to the created weaving model in order to automatically generate the needed transformations at the modeling level.

Once those transformations are available, a typical model transformation usage scenario is:

1. modeling the software architecture using a given ADL;

2. applying the DUALLY model transformations to the initial architecture model in order to obtain the architecture description in the target ADL;

3. working on the architecture description by using the target ADL.

Moreover, the weaving models layers makes DUALLY model transformation language independent. Indeed, software architects just have to create the needed weaving models and the DUALLY transformation engine automatically generates the needed model transformations at the modeling level. More concretely, the version of DUALLY which is described in this dissertation generates ATL model transformations (see Section 4.4); additionally, there is an experimental prototype of DUALLY which generates transformations conforming to a prototype of the Janus Transformation Language (JTL\(^5\)); please refer to [45] for further information on the JTL prototype of DUALLY.

In the following I describe how weaving models are used to define metamodel to metamodel links and the needed higher-order transformations to create interoperability among architecture models.

\(^5\)Janus Transformation Language website: http://jtl.di.univaq.it
3.2 DUALLY: a solution for providing architecture description languages interoperability

Metamodel to metamodel Links: Bringing metamodels together

What I earlier called metamodel-to-metamodel links are the main branches that stem out of $A_0$ and they constitute a mechanism to provide references and semantic relations among elements of the interoperating ADLs and $A_0$. As previously stated, weaving models define these links so as to later use them to generate lower level model transformations. These links are defined once, during the process that integrates a certain architectural language within DUALLY. Their role is in essence that of constituting the bridge between $A_0$ and the DUALLYzed architecture description language.

A weaving model (e.g., $wm1$ in Figure 3.8) defines a set of links between an ADL metamodel and $A_0$. A number of methods to specify and construct weaving models is currently being explored and exploited; conceptually, weaving models conform to a given weaving metamodel, and can be defined either manually or by utilizing ad-hoc scripting languages. The weaving metamodel provided by DUALLY defines the types of link that can be established in a weaving model. Its main elements are:

- directed and bidirectional correspondences to relate two or more concepts,
- feature equivalences to relate attributes or references,
- bindings to user-defined constants.

A complete description of the DUALLY weaving metamodel is provided in Section 4.4. The definition of the weaving model constitutes the main step to be carried out in order to integrate a certain architectural language into the languages topology. The weaving models (and so their corresponding model-to-model transformations) contain the rationale that enables the transformation engine to let two model instances interoperate. DUALLY enables the (graphical) definition of weaving models.

Higher-order transformations: Creating Lower Level Communication

As introduced in Section 2.2.2, a model transformation can be considered a model. Recent research efforts enhanced the concept of model transformation, promoting it into that of transformation model [9]. Just as a model can be created, modified, or refactored through a transformation, a transformation can be regarded as a model and therefore, it can itself be instanced, modified and so on.

Suitably constructed transformations can carry out this particular task: thanks to higher-order transformations specified and developed upon DUALLYzation, model-to-model transformations can be obtained. In DUALLY these transformations are automatically executed directly. Uniting all these transformations, a network is obtained: this will enable full, level-wide interoperability.

As in standard model transformations engines, the meta-concepts in ADL1 and $A_0$ (and the relations linking them) are defined once, whereas models to be transformed can change.
As can be noticed in Figure 3.8, each weaving model is automatically transformed into a model-to-model transformation that can be then executed by its underlying transformation engine. This is performed by means of the higher-order transformations that handle each element of the weaving metamodel and generate its corresponding model transformation code. Thus, the quality of the resulting model-to-model transformation is strongly affected by the way the weaving model is defined. Each relation between corresponding elements should be defined as precise as possible. Implementation details on how weaving models are managed in DUALLY will be provided in Section 4.4.

### 3.2.2 The $A_0$ Pivot Language

The main purpose of $A_0$ is to provide a centralized set of architectural elements with respect to which relations must be defined. Considering that many languages need to be related together (see Figure 3.6), instead of creating a point-to-point relationship among all languages, a linear relationship between the selected language and $A_0$ is created, thus reducing the number of connections needed.

$A_0$ is the base language that every ADL may use to keep itself “aligned” with any other integrated one. All the passages between ADLs are referenced through $A_0$ and this makes sure that architectural information is maintained consistent at every step of the cycle: modeling $\rightarrow$ transformation $\rightarrow$ modeling and so on.

The selection of the elements within $A_0$ was mainly guided by the principle of maintaining $A_0$ as general as possible to ensure that DUALLY is able to potentially represent and support any kind of architectural representation (e.g., formal ADLs, UML-based languages, etc.). $A_0$ has been defined as general as possible to ensure that DUALLY is able to potentially represent and support any kind of architectural representation (e.g., formal ADLs or UML-based languages). As shown in Figure 3.9, the selection phase has been performed by studying the state of the art on architecture description languages.

Moreover, in order to be sure that $A_0$ contains a representative set of state-of-the-art architecture description languages, its metamodel has been double checked during the survey I worked on about the usage of ADLs in industry (see Section 1.2). More specifically, in order to define a preliminary set of ADLs to be considered, I searched for papers published in main conferences, journals, and events on software architectures, and I contacted experts on software architectures. Among these ADLs we can find ACME [59], xADL [33], Rapide [97], Darwin [99], Wright [2], Koala [146], and MetaH [13]. In a second step I defined Google Scholar\(^6\) and Microsoft Academic Search\(^7\) queries in order to complete the set of ADLs identified in the first step. I chose Google Scholar and Microsoft Academic Search because they search both in the Internet and in the editorial online databases like ACM, Springer, and so on; in this way I could get both research papers, white papers, technical reports and other kinds of documents.

\(^6\)Google Scholar website: [http://scholar.google.com](http://scholar.google.com)

\(^7\)Microsoft Academic Search website: [http://academic.research.microsoft.com](http://academic.research.microsoft.com)
### 3.2 DUALLY: a solution for providing architecture description languages interoperability

Listing 3.4 shows an example of Microsoft Academic Search query. I looked only for documents published after 1991 because it is well-known that the concept of ADL did not exist before that year. This query performed on the Computer Science domain returned a list of 102 references\(^8\). Due to the manageable size of the produced results, there is not the need to refine further the query with the aim to filter out parts of the results.

\[
\text{ADL architecture description language year} \geq 1991
\]

**Listing 3.4: Example of Microsoft Academic Search query**

The same query used for querying Google Scholar produces a list of 10,800 hits\(^9\). Listing 3.5 shows a refinement of the query that produces a list of 2,730 hits.

\[
\text{ADL "architecture description language"}
\]

**Listing 3.5: Example of Google Scholar query**

Furthermore, I discovered that another community call ADLs languages for specifying the architecture for retargetable compilation [126]. Therefore, I refined the search query excluding results that contain names of approaches that fall in this other domain, such as EXPRESSION ADL, Valen C, etc. The query might be found in Listing 3.6. The query is executed by looking only for documents published after 1991 and restricted to the computer science domain.

\[
\text{ADL "architecture description language" - "EXPRESSION ADL" - "Valen C" - MIMOLA - nML - ARC - Axys - RADL - Tensilica - MDES - TDL - Flexware - HMDES - PRMDL - Maril - CSDL}
\]

**Listing 3.6: Refined Google Scholar query**

The execution of the Google Scholar query produces 1,930 hits as result. The list of 57 ADLs produced in the first step is enriched with the ADLs mentioned in scientific papers, white papers, and technical reports, which are discovered in step two. More precisely in step two we obtained 102 references by using Microsoft Academic Search, and 1,930 references by using Google Scholar. This process produces a list of 120 ADLs\(^10\), and for each of those ADLs I collected scientific papers, white papers, and technical reports. The output of this process is then used to build \(A_0\).

Basically, the main elements of \(A_0\) are Architecture, i.e., a collection of components (SAcomponent) and connectors (SAconnector) instantiated in a configuration, SAinterface which specifies the interaction point between an SAcomponent or an SAconnector and its environment, SAtypes to define architectural types, its specialization SAsructuredType which can contain also a definition of sub-architectures, and so on. Please refer to Section 4.4 for further details about \(A_0\) and its metamodel.

It is important to note that \(A_0\) has been designed as a minimalistic set of architectural concepts. I decided to do so because an approach with a minimalistic pivot language scales better. More specifically, Figure 3.10 shows the main difference between the use of

---

\(^8\) Query performed on July 4, 2011

\(^9\) The queries have been performed on July 4, 2011

\(^10\) The complete list of ADLs can be found at: http://goo.gl/e6vda
a rich pivot language and a minimalistic one: filled circles represent modeling concepts, solid lines denote correspondences among AL and pivot language concepts, and finally dashed boxes and dashed lines represent added ADLs and correspondences, respectively. A rich pivot language is built with the aim of including the highest number of concepts contemplated by all the interoperating ADLs. As shown in Figure 3.10.a, each concept in any ADL finds its correspondence with a rich pivot language element. Differently, a minimalistic pivot language contains only a core set of concepts (as shown in Figure 3.10.b), and is kept as small as possible.

Such a difference has positive and negative impacts on the scalability of the whole interoperability mechanism. Indeed, as soon as a new ADL has to be considered, an hypothetical rich version of $A_0$ could need to be revised in order to avoid information loss. As shown in Figure 3.10.a, the insertion of $ADL_4$ implies the addition of the link between $ADL_4$ and the already existing element $b_1$ in the rich pivot, and the addition of $b_2$. This may require a strong revision of the entire $A_0$ to solve possible conflicts and to avoid inconsistencies. On the contrary, being $A_0$ minimalistic, when $ADL_4$ is linked to the $A_0$ (see Figure 3.10.b) only a new correspondence with $b_1$ is created.
3.2 DULLY: a solution for providing architecture description languages
interoperability

Scalability is a very useful property for a pivot language, however it comes with a cost: low interoperability accuracy. More specifically, $A_0$, being minimalistic, may discard some correspondence, thus limiting the interoperability accuracy. For instance, see $a1$ and $a2$ in Figure 3.10.b: while a correspondence among them may be found in the rich pivot, it is missing in the solution with the minimalistic $A_0$. Information loss is thus introduced. The minimalistic approach may be particularly limiting when domain-specific ADLs are introduced in the star topology. A possible solution to interoperability inaccuracy consists in making the $A_0$ pivot language as extensible, thus adaptable to new architecture description languages.

**Extensibility mechanisms of $A_0$**

While designing the extensibility mechanism for **DULLY**, two main requirements have been taken into consideration:

1. The extension process must be **systematic**. For example, supposing that both real-time and behaviour extensions are needed. So far, three alternative solutions can be applied: i) extend $A_0$ with real-time concepts first, then with behaviour, ii) extend $A_0$ with behaviour concepts first, then with real-time ones, iii) extend $A_0$ with both concepts at the same time. The three scenarios may produce different kernel pivots, and so far there is no guideline on how to manage such a multiple extension. So, the extension process has to provide to software architects clear guidelines on how and what to extend.

2. The extension mechanism must be **compositional**. Indeed, current solutions tend to create ad-hoc extensions, not engineered to be reusable. Even when applying scenarios i) or ii) above, the intermediate extensions are typically lost and not stored for reuse. Moreover, the extension itself is not considered as a first class element, but simply as an improvement to the original pivot. This limits an engineered approach to language extensions. So, it is very much preferable to have a compositional and reuse-oriented extension mechanism where $A_0$ extensions can be re-used and extended.

In order to satisfy the above mentioned requirements, the $A_0$ has been ”promoted” from being a single metamodel into being a hierarchy of interrelated metamodels [37]. Both the various versions of $A_0$ and their related extensions (formalized as metamodels as well) are part of the $A_0$ hierarchy. Figure 3.11 shows how the $A_0$ hierarchy can be constructed and how the extensibility mechanism is able to overcome the problem of low interoperability accuracy.

The extensibility mechanism relies on the adoption of weaving models [7] which relate an extended $A_i$ with an extension $e_i$. A weaving model $wm$ contains links between elements of an extended language $A_i$ and elements of an extension $e_i$. The generation of an
extended language $A_e$, which is an extension of $A_i$ with $e_i$, is performed by executing a generic higher-order transformation, which is defined once forever. The transformation applies the extension $e_i$ to $A_i$ according to the extension operators used in $wm$. These operators will be described in Section 4.5.

Extension hierarchies, like the one in Figure 3.11, contain three types of elements: pivot languages, extensions, and weaving models that apply extensions to pivot languages. In order to regulate how pivot languages and extensions can be involved in specific weaving models, I make use of a type system for pivot languages and extensions. In other words, a weaving model defined for a pivot language can be re-used also for applying extensions to other pivot languages, under the assumption that these pivot languages have the metaclasses involved in the weaving model.

The extension mechanism, starting from the minimalistic $A_0$, (i) permits the automated construction of a hierarchy of pivot languages (blue bubbles in Figure 3.11), and (ii) provides mechanisms to transform from an ADL to another (i.e., ADL1 and ADL2 in figure) by minimizing the loss of information; under this perspective, the problem of having low interoperability accuracy is solved by applying the generated model transformations through the most informative pivot language in the hierarchy for the considered ADLs.

Once a suitable pivot languages hierarchy is constructed, ADLs can be bound to different pivot languages of the built hierarchy ($A_i$ in Figure 3.11). As discussed in Section 3.2.1, binding an ADL to a specific pivot language in the hierarchy means creating a weaving model between them. Languages extensibility is a wide research area so, in order to keep this dissertation well focussed I do not include here the details about the extensibility mechanism discussed in this section, the interested reader may refer to [37] for more details.

### 3.2.3 THE ACC CASE STUDY

In this section I describe a scenario in which DULLY has been applied in order to provide interoperability between three different architecture description languages. The first language is called SaveCCM [68] has been defined for the vehicular system domain. It
is used in the context of a domain specific component model, the SaveComp Component Model (SaveCCM) to support the development of resource-efficient systems. Systems in this domain are highly critical for the vehicles functionality and controlling, and both structural and behavioral aspects need to be carefully assessed before development. While SaveCCM allows for structural preventive analysis, a different tool is required for behavioral analysis. Similarly to what happen in the Save project\(^\text{11}\) [90], the verification features of LTSA (Labelled Transition System Analyser) [100] are exploited in order to increase the dependability of the product under development. For doing so, a specification in the Darwin/FSP ADL [101] is required\(^\text{12}\). While SaveCCM and Darwin/FSP provide domain- and analysis-specific notations and tools, a more commonly known description language may be required for sharing architectural knowledge among the different stakeholders involved in the project. For this purpose, a UML profile for component-based architectures called UML\(_{cc}\) [78], and UML tools documentation features, like the UML MagicDraw reports\(^\text{13}\) are used. This profile contains mechanisms to specify systems via components, connectors and their behaviour. The profile is described in details in Section 5.2.3.

![Figure 3.12: ACC modeling with DUALLY](image)

In this specific scenario, and without suitable technologies like DUALLY, software architects are asked to accurately model the system in SaveCCM, then to re-model the system from scratch in Darwin/FSP in order to be able to perform the desired analysis by means of LTSA. Once obtained a satisfactory model, they have also to re-model the system from scratch in a UML tool suitable for the UML\(_{cc}\) profile. In addition to the overhead required to re-model the system three different times, this scenario opens several potential problems. First of all, it is up to software architects to write “equivalent” models in each

\(^{11}\) Save project: [http://www.mrtc.mdh.se/save/](http://www.mrtc.mdh.se/save/).

\(^{12}\) It has to be noted that while FSP provides behavioral information not included in SaveCCM, Darwin contains structural information to be aligned to the SaveCCM model.

involved language and tool. Moreover, at each step models are refined and possible errors are discovered and solved, the propagation of changes requires by-hand re-modeling. Clearly, cost and time requirements prohibits the different models consistency.

Figure 3.12 presents an overview of the ACC scenario. For the sake of simplicity, in this scenario I assume that all the languages metamodels (i.e., the SaveCCM, UML and Darwin/FSP metamodel) have been already linked to DUALLY’s $A_0$ by means of three different weaving models and that the architect already generated the model transformations implementing the logic of those weaving models.

The preliminary step consists modelling the system in SaveCCM. The considered system is an Adaptive Cruise Controller (ACC) that has been a recurring example throughout the development of SaveCCT (e.g. [90]). ACC is an extension of a traditional vehicular Cruise Controller with also a set of sensors (mainly radars and cameras) that allow the controller to (i) check if there is another vehicle in the lane, allowing the driver to maintain a safe distance with the preceding vehicle and (ii) check the presence of speed road signs and adapt the actual speed according to them. Figure 3.13 shows the SaveCCM model of the ACC system.

![Figure 3.13: SaveCCM model of the ACC system](image)

The ACC system is composed of two main components: Sensors and ACC_Application. The former ones periodically provides data about the distance towards the preceding vehicle, actual speed, status of the whole controller and the status of the brake pedal; whereas the latter is the core of the system. It accomplishes three main tasks: (i) analyze information provided by sensors and plan actions to be performed by the actuators; (ii) log the status of the system; (iii) provide data to the driver display (Human Machine Interface - HMI). For the sake of simplicity, the internal structure of ACC_Application and the external context of the whole system are not described here. Three clock components regulate the timing of the ACC system.
3.2 DUALLY: a solution for providing architecture description languages interoperability

Once a satisfactory model has been designed, the whole process of the motivating example is organized into five main steps:

1. The first step consists in translating the initial model from SaveCCM to $A_0$. It is important to note that this step is automatic, thanks to the DUALLY model transformation generation capability; the architect has just to execute the transformation generated from its associated weaving model. Figure 3.14 shows the resulting $A_0$ model. Specifically, the clocks, the Sensors component and the ACC APPLICATION assembly become $A_0$ SComponents. Their ports become SAInterfaces independently from their nature (i.e. trigger or data). Finally, SaveCCM connections are transformed into SChannels.

2. The second step consists in translating the ACC model to both Darwin/FSP and UML_{cc}. Figure 3.15 and 3.16 show the generated models in Darwin/FSP and UML_{cc}, respectively. Those models are quite self-explaining, so I will not go
into the details on their elements. The architect will now focus on the Darwin/FSP model.

3. While the Darwin specification is obtained by transforming SaveCCM, FSP state machines need to be done by hand (since behavioral information is missing in SaveCCM). Listing 3.7 shows the FSP process representing the behavior of the ACC_Application behavior.

Listing 3.7: The ACC_Application behavior in FSP

```
1 ||ACC_Application = (COMPUTESTATUS || LOG HMI).
2 COMPUTESTATUS = (rule -> ACTIVE),
3 ACTIVE = (tick3 -> checkStatus -> dataAvailable ->
4 readDistanceData -> readSpeedData ->
5 readAcceleratorData -> readBrakeData -> COMPUTE
6 | off -> COMPUTESTATUS),
7 COMPUTE = (compute -> trigActions -> ACTIVE),
8 LOGHMI = (on -> ACTIVE),
9 ACTIVE = (tick1 -> provideHMIData -> log -> ACTIVE
10 | off -> LOGHMI).
```

The listing shows that the ACC_Application is organized into two different parallel processes: COMPUTE_STATUS and LOG_HMI. The COMPUTE_STATUS process encodes the retrieval of the data information (namely, readDistanceData, readSpeedData, readBrakeData, and readAcceleratorData) once a clock is received (tick3) and the data are made available by the Sensors component (dataAvailable). The LOG_HMI process is activated by clk10Hz (tick1), provides the data to the HMI and logs the status of the system.

The modelled behavior is verified by means of LTSA and the tool discovers an error when checking a specific property of the system. This analysis highlighted a problem of robustness of the current version of the ACC software architecture. In fact having two different clocks, even if synchronized at the same rate, can lead an architect to modify the rate of a clock without accordingly adapting the other
one. Therefore, the ACC model has been refined in Darwin/FSP by introducing a new component, *Splitter* in Figure 3.17: it receives a message from a single clock and splits it into two messages: one is sent to *ACC Application* and the other one to *Sensors*. This prevents the architect to break the temporal alignment of the two components.

Figure 3.17: Revised Darwin model of the ACC system

4. At this point a change propagation back to $A_0$ is now required, so the changes made in Darwin/FSP can be propagated to $A_0$, and then to SaveCCM and UML$_{cc}$. Figure 3.18 shows the revised model in $A_0$ that contains all the changes made in Darwin/FSP. In the figure can be noted the addition of the *Splitter* connector and the behaviour specification of the system.

Figure 3.18: Revised Darwin model of the ACC system

Figure 3.19 and Figure 3.20 show the revised versions of the ACC system in SaveCCM and UML$_{cc}$, respectively.
Figure 3.19: Revised SaveCCM model of the ACC system

Figure 3.20: Revised UML\textsubscript{ACC} model of the ACC system
3.3 BYADL: a solution for extending and customizing architectural languages

It is important to note also that, since the revised Darwin/FSP model of the ACC system contains also behavioural information, state machines corresponding to the behaviour of the system have been automatically generated when transforming to UMLc.cc.

Figure 3.21: Behaviour of the ACC Application component as UML state machines

Figure 3.21 shows the behaviour of the ACC Application obtained by transforming the behaviour in $A_0$ and originally added in Darwin/FSP. As can be seen in figure it is organized as two parallel UML state machines. The obtained UMLc.cc model can then be used to generate the required documentation of the system as MagicDraw browsable reports.

3.3 BYADL: A SOLUTION FOR EXTENDING AND CUSTOMIZING ARCHITECTURAL LANGUAGES

BYADL [36] is an approach for extending and customizing existing ADLs according to domain- and organization-specific concerns (e.g., domain-specific concepts, analysis features). Figure 3.22 gives an idea of how an ADL could be extended and customized via BYADL.

Conceptually, the BYADL composition engine allows to extend the ADL with domain-specific concerns, with new architectural views, with analysis constructs or with methodology and process concepts (see lower part in Figure 3.22), depending on the system’s stakeholder concerns. The ADL could be also customized; this is done by composing it with ad-hoc metamodels (see left upper part in Figure 3.22) representing only the customization aspects of the language (i.e., only the entities that change or must be added).
Fundamentally, BYADL provides an incremental approach to build customized and customizable ADLs starting from an already existing one (ADLx in the figure); the new ADL is build by means of model-driven extension mechanisms.

It should be noted that extensions in BYADL are defined in an ADL-independent manner and they are collected in the MEGAF megamodel so that they can be reused for further extensions of (even different) ADLs. Finally, it is also important to highlight that the approach promoted by BYADL is incremental, so that software architects are able to extend and customize their ADL whenever required. This is particularly important in practice since often the characteristics of ”optimal” ADL (for instance as required by the considered non-functional aspects) may change during the architecting phase [120].

Figure 3.23 presents an high-level overview on the functioning of BYADL. The architecture description language obtained at the end of the process consists of four main elements:

1. abstract syntax: the metamodel obtained by means of the BYADL composition mechanisms (\( M_{comp} \) in Figure 3.23);
2. concrete syntaxes: a set of concrete representations that define the way architecture models look like to the software architects;
3. semantics: it describes the meaning of each construct of the extended language.
4. model migrators: specific model transformations that are able to reflect the models defined within the newly extended ADL, back to the original tools.
In the next sections I provide more details on how BYADL manages each main element described above.

### 3.3.1 Abstract Syntax of the Extended ADL

The abstract syntax of the newly extended ADL is the metamodel obtained by means of the BYADL composition mechanisms. The starting point for obtaining such a metamodel is represented by the metamodel of the ADL to be extended; it is contained into the MEGAF megamodel, together with other metamodels representing other architectural concepts, like domain-specific elements, analysis constructs, views, methodologies, and so on. The metamodel of $ADL_x$ is extended by applying specific metamodel composition operators. Each operator is always applied on two metaclasses (one belonging to $MM_{ADL_x}$ and one to $MM_{ext}$) that I refer to as source ($s$) and target ($t$) in the remainder of this section. The extension operators are:

- **Match**: $s$ and $t$ represent the same concept; they are merged into a single metaclass which contains the union of all the structural features (i.e., both attributes and references) of $s$ and $t$. Their supertype and subtype references are merged as well.

- **Inherit**: this operator specifies that the concept $s$ will be a subtype of $t$ in the resulting extended metamodel. If its application results in a cycle in the inheritance tree, then it is not executed and a warning is raised.
• **Reference**: in the extended kernel, \( s \) has a reference to \( t \). All features of the new reference (e.g., lower-bound, name) can be set while applying this operator.

• **Expand**: all the attributes of \( s \) are copied into \( t \). Attributes with the same name are merged.

Figure 3.24: Generation of the abstract syntax of the extended ADL

A detailed description of the composition operators and how they have been formalized is given in Section 4.5.1. Figure 3.24 shows an overview on how \( bYADL \) generates the abstract syntax of the extended/customized architecture description language.

The composition operators are applied by executing an higher-order transformation \( T_{\text{comp}} \) that takes as input three models:

- \( MM_{\text{ADLx}} \): the metamodel of the ADL to be extended,
- \( MM_{\text{ext}} \): the metamodel containing the concepts that will extend the resulting ADL, and
- \( WM_{\text{comp}} \): the weaving model containing the application of the composition operators to the metaclasses in \( MM_{\text{ADLx}} \) and \( MM_{\text{ext}} \)

\( T_{\text{comp}} \) implements the logic of the operators applied in the weaving model \( WM_{\text{comp}} \) and produces as output the metamodel of the resulting extended ADL (\( MM_{\text{comp}} \) in Figure 3.23). Section 4.5 presents in details this aspect of \( bYADL \).
3.3 BYADL: a solution for extending and customizing architectural languages

3.3.2 Concrete Syntaxes of the Extended ADL

BYADL allows to automatically generate different notations to visualize and edit architecture models. There are three different possibilities to produce an editor for the ADL being developed in BYADL: tree-based, textual, and graphical.

The tree-based editor, with its collapsible and hierarchical structure, is automatically provided by means of the Eclipse Modeling Framework (EMF\(^{14}\)), the underlying technological solution for managing the models within MEGAF (it will be described in Section sec:technologies. It displays model elements through a collapsible and hierarchically structured tree with content outline and property sheets. If needed, the tree-based editor can be customized either by hand-coding in Java the specificities of the editor, or by using specific Eclipse-based technologies.

The textual editor is automatically generated and conforms to the standard Human-Usable Textual Notation (HUTN) specification\(^{15}\). The produced textual editor supports syntax highlighting and automatic conformance check with respect to the metamodel of the extended ADL. Ad-hoc concrete syntaxes may be also used for the new ADL, however this aspect is not automatizable: in this case the software engineer needs to specify what is the textual representation of each element of the ADL metamodel.

The graphical editor is based on the EuGENia\(^{16}\) tool: exploiting specific annotations of the metamodels involved in the composition, a graphical editor is automatically generated. Obviously the generation of the editor is limited to elements for which suitable annotations are provided. Special policies regulate the choice of the graphical element to be used when more than one metamodel provides annotations for a specific concept.

Each editor is characterized by different levels of usability and requires different effort for the customization (if needed). A thorough discussion of this feature is given in Section 4.5.

3.3.3 Semantics of the Extended ADL

The semantics of the extended ADL is given by means of semantic relationships between the language’s elements and elements of the \(A_0\) metamodel (see Section 3.2). By means of such relationships, the elements of the ADL implicitly inherit the semantics of \(A_0\).

Once a software architect has provided relationships between elements of an ADL metamodel and elements of \(A_0\), the elements of the composed ADL (represented as the \(MM_{\text{comp}}\) metamodel) will be automatically linked to \(A_0\). In so doing, the composed ADL implicitly inherits the built-in semantics of \(A_0\). Note that this way of providing semantics is

\(^{14}\text{EMF website: http://eclipse.org/emf/}.

\(^{15}\text{HUTN specification: http://www.omg.org/spec/HUTN/}.

\(^{16}\text{EuGENia: http://www.eclipse.org/gmt/epsilon/doc/eugenia/}.\)
(conceptually) the same proposed in [132] in which the semantics of a Domain Specific Language (DSL), called translational semantics, is provided by means of model-to-model transformations from the DSL to another language. In my case the target language is $A_0$. For this reason BYADL asks the software architect to define these semantic relationships between the ADL to be extended and $A_0$, and between the metamodel that represents the extension and $A_0$ as a pre-requisite to operate with BYADL. This activity is performed at the time in which those metamodels (i.e., the one of the ADL and the one representing the extension) are imported into the MEGAF megamodel. Basically, for each element of the ADL and of the metamodel that represents the extension, the software architect selects, through the graphical interface of MEGAF, the most appropriate metaclass of $A_0$. The result is that each element of both the ADL or the extension metamodel has one and only one type-of [41] relationship with an element of $A_0$. For instance, a Darwin component instance is typed $SAComponent$, a Darwin component declaration is typed $SAStructuredType$, and a Darwin portal is typed $SARole$, and so on. Please refer to Section 4.2.3 for further details on how MEGAF manages semantic links between the metamodels in the megamodel and $A_0$.

Section 4.5.3 presents in details how BYADL allows to generate semantic links between the composed metamodel $MM_{comp}$ and $A_0$. There, I discuss the importance of the semantic links to $A_0$ by describing how the semantic links help architects during the composition phase and improve the quality of the concrete syntaxes of the composed ADL.

3.3.4 MODEL MIGRATORS

In BYADL specific model transformations are able to reflect the models defined within the newly extended ADL, back to the original tools. These transformations are automatically generated by higher-order transformations and are fundamental for analysing architecture models using their original tools. In this dissertation I will refer to this transformations
3.3 BYADL: a solution for extending and customizing architectural languages

as model migrators, in order to better characterize their tool-tool migration objective. So, model migrators are specific model transformations that allow to pass from models conforming to the composed metamodel to models conforming to the initial ones, and vice versa.

By referring to Figure 3.23, we can suppose that the software architect is composing $\text{MM}_{\text{ADLx}}$ and $\text{MM}_{\text{ext}}$ in order to obtain the composed metamodel $\text{MM}_{\text{comp}}$. Two kinds of model migrators may be generated by BYADL:

1. **Composed2SingleMigrator** takes as input a model conforming to $\text{MM}_{\text{comp}}$ and produces a model conforming to $\text{MM}_{\text{ADLx}}$ and a model conforming to $\text{MM}_{\text{ext}}$. Figure 3.26 shows the **Composed2SingleMigrator** migrator.

![Composed2SingleMigrator](image)

**Figure 3.26**: Composed2SingleMigrator model migrator in BYADL.

2. **Single2ComposedMigrator** transforms in the inverse direction of the other migrator; i.e., it takes as input two models conforming to $\text{MM}_{\text{ADLx}}$ and $\text{MM}_{\text{ext}}$ respectively and transforms them into a model conforming to $\text{MM}_{\text{comp}}$. Figure 3.27 shows the **Single2ComposedMigrator** migrator.

![Single2ComposedMigrator](image)

**Figure 3.27**: Single2ComposedMigrator model migrator in BYADL.
Chapter 3. Solution

The inner logic of each migrator is represented by the operators applied in the weaving model during the composition phase (i.e., $W_{M_{comp}}$). Model migrators may be used also outside the BYADL toolset. The migrators generation engine of BYADL manages also trace models produced during the execution of the Single2ComposedMigrator in order to keep track of how two separated models must be recomposed. Section 4.5 presents in details how migrators can be automatically generated.

3.3.5 The IECS Case Study

In this section I describe an application of BYADL in which it has been applied to extend the Darwin ADL with concepts coming from a UML profile, BPMN and a custom metamodel. More specifically, supposing that company X has invested time and effort for acquiring the required knowledge and experience on the Darwin ADL [99], and would like to use such an ADL as much extensively as possible. Supposing also that a fault-tolerant system must be modelled where it is extremely important to explicitly model (i) software connectors, (ii) both normal and exceptional behaviors, and (iii) the development strategies related to the SA, unfortunately, all those features are not supported by Darwin. This case study is organized in three scenarios that permit to show how BYADL can be used incrementally and how the various composition operators can be applied to extend Darwin with all the features covering the above mentioned concerns. Figure 3.28 shows how the three scenarios have been organized.

The first step is to consider the Darwin ADL as the starting point for the composition. Then, the concerns of the current system being developed (e.g., fault tolerance, direct link to the development process, and so on) will be covered by composing Darwin with other concern-specific concepts.
3.3 BYADL: a solution for extending and customizing architectural languages

**DarwinFT**: extending Darwin with Fault Tolerance. In order to generate DarwinFT, the Darwin ADL is composed with the *IdealComponent* UML profile presented in [38] for specifying software components according to the idealized fault tolerant component model [50]. Each ideal component is composed of both normal and exceptional parts and exceptions can either be signalled or handled through specific interfaces. Figure 3.29 show the mwA composition model that composes Darwin and IdealComponent to generate the DarwinFT metamodel.

![DarwinFT composition model](image)

**Figure 3.29: DarwinFT: Extending Darwin with Fault Tolerance**

The mwA composition model contains the application of the *inherit* and the *expand* operators. Each *inherit* operator has the exclusive property set to true. The *inherit* operator allows the specialization of the Portal metaclass of Darwin with HandlerInterface and RaiserInterface of the idealized component model; they are two interfaces specialized for managing and raising exceptions, respectively. Furthermore, the *inherit* operator is used to specialize a ComponentInstance of Darwin in NormalComponent and ExceptionComponent representing the normal and the exceptional part of a component, respectively. Finally, the *expand* operator is used to add the hasException attribute to a ComponentDeclaration of Darwin. Such attribute specifies whether the component’s behaviour is specified through the normal and exception sub-components.
**DarwinFT+BPMN**: DarwinFT & Development process in BPMN. In this scenario, the DarwinFT metamodel resulting from the previous step can be composed with the BPMN metamodel\(^\text{17}\). Upon doing so, software architects can associate structural parts of the system to specific development activities. The design/development process is directly linked to its business context, including development strategies, costs, risks, etc. Figure 3.30 show the \(mwB\) composition model that composes DarwinFT and BPMN to generate the *DarwinFT+BPMN* metamodel.

![DarwinFT+BPMN Diagram](image)

**Figure 3.30: DarwinFT+BPMN: DarwinFT & Development process in BPMN**

The \(mwB\) composition model makes an extensive use of the *inherit* operator. The main idea is to allow architects to associate Darwin structural elements (e.g., components, portals, etc.) to BPMN activities (e.g., tasks, sub-processes). In BPMN the Artifact element has been conceived as an extension point to provide additional information to BPMN processes, so each DarwinFT element extends the Artifact metaclass through the *inherit* operator.

\(^{17}\)BPMN specification: [http://www.bpmn.org/](http://www.bpmn.org/)
3.3 BYADL: a solution for extending and customizing architectural languages

(DarwinFT+BPMN)$_{cc}$: Darwin customization. In this step the DarwinFT+BPMN metamodel is customized by adding software connectors as first-class elements. The metamodel used for this communication is called softwareConnectorsMM and it has been built ad-hoc for this specific composition step (it can also be used for future extensions). Components may communicate also through connectors now and connectors have associated portals that act as architectural roles. Figure 3.31 show the mwC composition model that composes DarwinFT+BPMN to generate the (DarwinFT+BPMN)$_{cc}$ metamodel.

![Diagram](image_url)

Figure 3.31: (DarwinFT+BPMN)$_{cc}$: Darwin customization

This scenario makes use of the match operator with the aim to match the ComponentInstance metaclass of Darwin with the ComponentInstance of the softwareConnectorsMM metamodel. The result is a unique metamodel in which ConnectorInstance specializes ComponentInstance, and ConnectorDeclaration specializes ComponentDeclaration. It should be noted that the OCL constraint will also be part of the final composed metamodel; this avoids having connector instances declared as components and vice versa.
Once the desired \((\text{DarwinFT+BPMN})_{cc}\) metamodel is available, \textsc{byadl} allows the architect to generate a set of concrete syntaxes for such a metamodel. In order to show the possibilities and flexibility provided by \textsc{byadl}, in this scenario the tree-based, textual and graphical concrete syntaxes will be used. Figure 3.32 shows the tree-based and textual editors at work, and Figure 3.33 shows the graphical one. \textsc{byadl}, by means of its graphical interface, allows the software architect to select the most suitable solution. For instance in Figure 3.33 Darwin, the idealized component model and the connector model have been integrated into a single view (white area in the figure), while BPMN has kept its separated view (gray area in the figure). I added by hand the links between these two different views to graphically show in the figure the existent relationships among elements of these views.

At this point of the IECS scenario, the \((\text{DarwinFT+BPMN})_{cc}\) language is ready to be used by the architects, so now I give an overview on how the IECS system has been modelled by using \((\text{DarwinFT+BPMN})_{cc}\). The modeled system, called Integrated Environment for Communication on Ship (IECS) [106], is based on a specification coming from a project developed within Selex Communications, a company mainly operating in the naval communication domain. Fundamentally, IECS is a multi-tier environment capable of maintaining a fail-safe communication within a military vessel. The main functionalities of the system are: (i) provide several communication modes; (ii) manage and distribute operative messages; (iii) manage and configure transmission and reception over radio channel.
(iv) remote control and monitoring of the system; (v) implement communication security techniques. The purpose of this scenario is more about showing how the newly created \((\text{DarwinFT+BPMN})_{cc}\) language can be used in practice. Therefore, I do not provide further informations on the IECS system.

![IECS graphical model in BYADL](image)

Figure 3.33: IECS graphical model in \textit{BYADL}

The integration of Darwin with BPMN allows architects to model not only the software architecture of IECS, but also the adopted development strategies (see Figure 3.33). The IECS software architecture is composed of the \textit{Equipment}, \textit{Workstation}, \textit{Proxy}, \textit{DB} and \textit{CTSM} components. The type of the latter one (\textit{CTSMType} in figure) is defined using the idealized fault tolerant component model integrated in Darwin. \textit{Network} is a software connector; this is possible thanks to the customization of Darwin (third step of the scenario scenario). As can be seen in Figure 3.33, Darwin and BPMN are represented via two different views. \((\text{DarwinFT+BPMN})_{cc}\) enables to assign software components of the designed SA to developer teams. In particular, assuming that the company has two developer teams (namely, \textit{A} and \textit{B}), a system engineering team and a testing team, the dotted lines graphically show how Darwin components are associated to the BPMN tasks and pools describing the activities assigned to each team. More specifically, \textit{Proxy}, \textit{DB}, and \textit{Network} are assigned to the system engineering team, \textit{Equipment} and \textit{CTSM} to the development team \textit{A}, and \textit{Workstation} to the development team \textit{B}). Even if these relationships are not graphically rendered, they exist in the IECS model and can be accessed through
the Properties panel of both the graphical editors.

It is important to note that the current IECS model can be manipulated by the original Darwin tool by applying a chain of migrators generated during each step of the composition phase (see Figure 3.28); finally, a model-to-text transformation (developed in [106]) completes the bridge towards the Darwin tool.
In this chapter I will give the details on the realization of the proposed approach. Here, the main objectives are: (i) to provide evidence that the solution proposed in my research is implementable, (ii) to give an architecture-level system/tool description of the proposed solution, (iii) to describe tool support and its UI, and (iv) to provide an analysis of implementation problems I faced and their solutions. In summary, this chapter is purely technical and mostly related to model-driven engineering advanced techniques (like model weaving, higher-order transformations, etc.) and tool support. The various decisions made on the conceptual side and the rationale justifying them have been already discussed in Chapter 3.

Figure 4.1: Software architecture of the proposed solution

Figure 4.1 gives a high-level overview of the software architecture of the proposed approach. Mainly, MEGAF can be considered as the main solution for modeling architectures by means of reuse, composition and customization; in the context of MEGAF, DUALLly and byADL represent two dedicated sub-solutions for tackling the architecture models consistency and ADLs adaptation problems, respectively. The various integration points of MEGAF with respect to DUALLly and byADL are described in the next
sections of this chapter. It is important to note that MEGAF can be considered as a generic means to precisely define SA-related concepts, and thus it can be a good candidate for creating and developing other engines that may reuse the knowledge base provided by MEGAF.

The current prototype of MEGAF (and thus also DUALLY and BYADL) is released under the GNU General Public License (GPL) and it is built as an Eclipse plugin extending AMMA [11], a model management platform designed and developed by the AtlanMod Team, INRIA. In the next section I will start from the lower part of Figure 4.1 by giving some information about Eclipse, AMMA, and the other technologies I use for developing the solutions provided in this dissertation. Then, Section 4.2 will describe how to import ADLs, metamodels and architecture models in the current prototype of the proposed approach. The remaining sections of this chapter will deal with technical aspects of the approach in the context of MEGAF, DUALLY, and BYADL.

4.1 TECHNOLOGICAL OVERVIEW

Eclipse\(^1\) is an open source development platform comprised of extensible frameworks and tools for building, deploying and managing software across the life cycle. The Eclipse open source community has over 60 open source projects, ranging from modeling environments, code generation frameworks, development environment for various programming languages, etc. A unique aspect of the Eclipse community and the role of the Eclipse Foundation is the active marketing and promotion of Eclipse-based projects and wider Eclipse ecosystem. One of the strong points of Eclipse is its plug-in architecture, that allows software developers (and vendors) all over the world to extend the platform and contribute to the Eclipse community with their own plugins. Indeed, tens of projects are part of a common ecosystem that comprise various different extensions to the Eclipse platform, each of them with a specific focus and purpose. Eclipse plugins are defined as a combination of Java classes and XML-based configuration files.

One of these projects is Eclipse Modeling Framework (EMF)\(^2\) that is a Java framework and code generation facility for building tools and other applications based on a structured data model. Basically, EMF helps developers in rapidly turning models into efficient, correct, and easily customizable Java code. EMF is an implementation of the MOF specification and can be thought of as a highly efficient Java implementation of a core subset of the MOF API. However, to avoid any confusion, the MOF-like core metamodel in EMF is called Ecore. EMF provides several benefits like, among the others:

- model change notification,
- runtime model support,

\(^1\)Eclipse project Web site: www.eclipse.org.
\(^2\)EMF project Web site: http://www.eclipse.org/modeling/emf/.
4.1 Technological overview

- a persistence layer based on the standard XML Metadata Interchange (XMI \(^3\)),
- runtime model validation,
- a generic efficient reflective API for manipulating EMF objects,
- a generic user interface for viewing and editing models
- native interoperability with other EMF-based tools and applications.

The last point about interoperability with other EMF-based tools is fundamental for my research since it allows the MEGA prototype to directly utilize (architecture) models developed with external tools. More specifically, in order to have a homogeneous treatment of the architecture models, I am assuming that all architecture models within MEGA are EMF models and that each model conforms to its Ecore metamodel. I see this assumption as reasonable considering the benefits it offers; in the (few) cases in which some kind of metamodel is not available for the ADL being considered, a preliminary transformation step is needed to bridge from the specific format in which the ADL is encoded to its corresponding metamodel. This aspect is further discussed in Section 4.2.

AMMA is a model management platform designed and developed by the ATLAS Team at INRIA Institute. AMMA is built on top of EMF and then models and metamodels are integrated into the same platform with several modeling technologies, such as Ecore and UML2 (which is an EMF-based implementation of the UML 2.x OMG meta-model for the Eclipse platform). I selected AMMA since this model management architecture is extensible, it provides a usable and reliable model transformation language, and because it supports the concepts of weaving model and metamodel independence. Its core elements are:

- **Atlas Transformation Language (ATL)** [84] which is a QVT (Query/View/Transformation)-like\(^4\) model transformation language, with its own abstract syntax and environment (i.e., an execution virtual machine and IDE). It allows to transform a source model into a target model. ATL is a declarative and imperative hybrid language. This is one of the key features of ATL: it takes advantage of the simplicity of declarative constructs, while complex tasks (e.g., manipulating UML profiles) can be performed in an imperative fashion. The transformation is itself a model conforming to a specific metamodel. This permits the creation of higher-order transformations, i.e., transformations that produce ATL transformations. This feature allows DUALLY and BYADL to automatically generate ATL transformations as needed.

- **Atlas Model Weaver (AMW)** [39] is the platform that manages weaving models. It allows the definition of correspondences among models (or meta-models) and to

---

\(^3\)http://www.omg.org/spec/XMI/.

Chapter 4. Realization of the solution

establish semantic links among model elements. The links are saved in a weaving model, which conforms to an extensible weaving metamodel. The weaving metamodel input format is KM3 (Kernel Meta-Meta Model) [83], a language that provides a textual concrete syntax for the coding of meta-models in a Java fashion. The weaving and woven models are defined by the XMI specification. Besides, AMW is featured with a set of extension points that enable developers to add specific semantics to the weaving mechanisms.

- **AM3 (ATLAS MegaModel Management)** is a global resource manager that operates in a model-engineering environment. This component provides support for modeling in the large i.e., dealing with models, metamodels, tools, services and their relations as a whole, while ignoring internal details. Within one platform (local or global), a megamodel [12] records all available resources and acts as an MDE repository. From a practical point of view AM3 manages megamodel elements (for example ATL transformations, tools, UML models, metamodels) and provides user interfaces to manipulate them.

- Other peripheral tools are grouped into the ATP (ATLAS Technical Projectors) component. These tools perform model transformation tasks. A subset of ATP tools consists of injectors and extractors to/from other technical spaces (e.g., UML, ATL, XML, SQL etc...) with little or no loss in information.

4.2 Importing ADLs and other Architectural Artifacts into MegAF

In order to have a homogeneous modeling environment I make the assumption that all of the heterogeneous artifacts involved in the architecting phase are EMF models and that each model conforms to its Ecore metamodel. This enables the management of complex architectural artifacts (e.g., ADLs) since their complexity is defined and encoded in the metamodel, thus enabling programmatic management of (even complex) models. The assumption also seems reasonable in light of the recent Doc2Model (Document to Model) Eclipse project for parsing structured documents to produce EMF models. I see this assumption as reasonable considering the benefits it offers.

Thus, in order to manage an architectural artifact within MegAF, its corresponding metamodel must be **imported** into the MegAF megamodel. Without loss of generality, in this section I will deal with the various issues that may arise when importing ADLs; other architectural artifacts (such as extensions definitions for BYADL, auxiliary models, etc.) are managed in the same way as for ADLs. Importing an ADL into the MegAF megamodel is a three-steps process:

1. Obtaining the metamodel,
4.2 Importing ADLs and other architectural artifacts into MEGAF

2. Tagging a metamodel,

3. Providing semantics to a metamodel.

The following sections will describe each step of the MEGAF metamodel importing process.

4.2.1 Obtaining the Metamodel

The MEGAF tool prototype works on Ecore metamodels. If the Ecore metamodel of an architecture description language is not available, many techniques exist to obtain such a metamodel, depending on the kind of artifacts describing the language is available:

- the ADL is text-based: the architect needs to define a metamodel for the ADL and bridge such a metamodel with the grammar of the ADL by using technologies like TCS\(^6\), Xtext\(^7\), Gra2MoL\(^8\), and so on;

- a metamodel exists, but it is not based on EMF: MEGAF relies on the importing facilities provided by EMF (e.g., the ones for XML-based representations, metamodels conforming to MOF, and so on;

- the ADL is based on a UML profile: this is a special case since a quite relevant number of ADLs has been specified as a UML profile. In this case I developed a dedicated approach for automatically bridging between UML profiles and Ecore metamodels. The bridge is transparent to the architect since it autonomously operates both on UML profiles and all the involved UML models. The bridge is realized through model transformation techniques in the Eclipse platform. The approach is described in [107].

4.2.2 Tagging a Metamodel

Assuming that the Ecore metamodel of the considered notation is available, the software architect has to “tag” it to specify additional information about the nature of the elements it contains.

Tagging metamodels guides ADLs extensions in BYADL and keeps the involved metamodels organized. In fact, the BYADL metamodel composition operators are metamodel-independent and this could lead to undesired scenarios if not properly supported. For

---

\(^6\)TCS: http://wiki.eclipse.org/TCS
\(^7\)Xtext: http://www.xtext.org
\(^8\)Gra2MoL: http://modelum.es/gra2mol/
instance, as previously said the composition and merging of two different ADLs is not ac-
cepted in BYADL and by means of the tagging feature it prevent this kind of composition
for each metamodel tagged as ADL.

The tags available reflect the different kinds of metamodel involved in both DUALLY
and BYADL usage scenarios, namely:

- **ADL**: the metamodel represents the constructs of a specific ADL. For example, the
  DiaSpec metamodel described in Section 5.2.3 contains all the constructs that may
  be used within a DiaSpec specification [20]. Composed ADLs obtained by means
  of BYADL are automatically tagged as ADL.

- **Pivot**: the metamodel represents $A_0$ or one of its extensions (see [37]).

- **Extension**: the metamodel represents a set of concepts that may be used as an ex-
tension in the BYADL composition mechanism. Metamodels tagged as Extension
may contain the relations between concepts within a domain and their semantics
(this is the classical case when a Domain-Specific Modeling Language (DSML)
[89] is imported into MEGAF), the concepts related to a specific analysis technique,
the concepts resulting from the study of an architectural viewpoint, process-specific
concepts, such as developers activities, or concepts related to other phases in the
development life cycle (e.g., requirements, test cases, deployment configurations,
etc.).

- **Customization**: this kind of metamodel can be seen as a special case of Extension
  in which the metamodel represents the concepts for customizing a given ADL.

Figure 4.2 shows a fragment of the MEGAF metamegamodel (see Section 3.1.1 related to
the representation of tags within MEGAF.

![Figure 4.2: The concept of Tag in the MEGAF metamegamodel](image)

Tags are represented by the Tag metaclass which contains a name attribute representing
the identifier of the tag. The importing wizard allows the user to associate a set of tags
to the metamodel being imported, which will be an instance of the Metamodel metaclass
or one of its subclasses. An architect can associated one of the predefined tags described
above (e.g., ADL, Pivot, Extension, or Customization), depending on the concepts con-
tained in the metamodel under consideration and on the various activities it will undergo
4.2 Importing ADLs and other architectural artifacts into MEGAF

during while architecting. Furthermore, architects can also define additionally tags and associate metamodels to them; in this case the tagging activity turns out to be useful for keeping metamodels semantically organized and to better classify them depending on their concepts and purposes.

4.2.3 PROVIDING SEMANTICS TO A METAMODEL

The third step of the importing process is that of providing semantics to the metamodel being imported. As stated in Section 3.3.3, a kind of translational semantics is provided to a metamodel by linking it to the $A_0$ pivot language. By doing this, the elements of the metamodel implicitly inherit the built-in semantics of $A_0$. The links among the metamodel elements and $A_0$ are stored in a weaving model type-of links.

In BYADL, I extended the AMW interface so that the semantic links to the $A_0$ metamodel guide the application of the BYADL composition operators. More precisely, once applying an operator, the BYADL tool highlights as target the metaclasses that are semantically compatible with the source metaclass. In this respect, semantic compatibility of metaclasses depends on the type of operator being applied; for example, the Inherit operator may be applied only between two metaclasses (e.g. $A$ and $B$) that are linked to the same metaclass in $A_0$, or if $A$ is linked to a sub-class of the $A_0$ metaclass to which $B$ is linked. This kind of mechanisms help the definition of semantically sound metamodel compositions.

Furthermore, if the metamodel being imported is tagged as ADL and such a metamodel is already integrated with the DULLY star topology of ADLs, then a weaving model linking the metamodel to $A_0$ already exists. In this case, BYADL reuses such a weaving model and analyzes it in order to provide semantics to the metamodel. BYADL analyzes each weaving link between the ADL metamodel and $A_0$ and applies the following strategy:

- If the woven element $e$ is linked to a single $A_0$ element, then the $A_0$ element gives the semantics to $e$;
- If the woven element $e$ is linked to more than one $A_0$ element (e.g., $A$ and $B$), then BYADL navigates the $A_0$ metamodel until to the most specific common super-class of $A$ and $B$. Such a metaclass gives the semantics to $e$;
- If there is no weaving link to/from the element $e$, then it is linked to the Element metaclass (i.e., the root of the $A_0$ metamodel).

Linking the metamodel to $A_0$ is not always mandatory, the tagging step helps in forcing the software architect to semantically link the metamodel only when needed. More specifically, if the metamodel is tagged either as ADL or Extension, then the software architect
should link it to $A_0$, otherwise the semantic linking is just optional. Indeed, the semantic links between an ADL metamodel and $A_0$ strongly help software architects while using BYADL, specifically: they guide the composition phase and improve the quality of the concrete syntaxes of the composed ADL.

### 4.3 Realization of MegAF

As already anticipated in the previous section, the current prototype of MegAF is implemented in the context of the AMMA platform. More specifically, it is defined as an extension of the AM3 component of AMMA. This extension is composed of three main elements:

1. the GMM4SA metamegamodel (see Section 3.1);
2. a set of Java classes that extend the core AM3 viewers and implement some auxiliary mechanisms of MegAF;
3. a set of OCL and ATL specifications implementing the consistency and completeness checks for megamodels conforming to GMM4SA.

Some functionalities of MegAF are inherited from the AM3 engine and from its available extensions; they are presented in the next subsection. In order to provide software architecture-specific functionalities, a set of extensions to the core AM3 plugins of the Eclipse and AMMA platforms are needed; the added functionalities are later described in a dedicated section.

### Basic Tool Support

By extending its basic metamegamodel, AM3 (together with its extensions) provides a set of built-in features out-of-the-box. First of all, the generic Eclipse Modeling Framework (EMF) provides a basic editor for creating megamodels conforming to the GMM4SA metamegamodel; the editor is based on a generic, tree-based view, an outline view and a properties view in which attributes and references of the megamodel elements can be set. In such an editor, model-specific properties and meta-data (key/value pairs) may be associated to each element of the megamodel (e.g. architecture framework, architecture description, stakeholders, concerns, and so on). Meta-data is useful when the software architect needs to associate system-specific information to some element of the megamodel.

The core AM3 engine also provides basic tool support for interacting with megamodels. Figure 4.3 shows its main components by means of the SBSCS example described in Section 3.1.2. A generic megamodel navigator allows to browse the current megamodel.
4.3 Realization of MEGAF

It is divided into two main viewers: the first viewer (shown in Figure 4.3.a) represents the inheritance tree of the metamegamodel and allows architects to navigate across the various element-types of the megamodel. The second viewer (shown in Figure 4.3.b) is used to display, once a user clicks on one element-type, all the megamodel elements which conform to the selected element-type. Then, each megamodel element can be edited by double-clicking on it. The editor for the currently selected megamodel element is shown in Figure 4.3.c. I extended the basic AM3 megamodel element editor to consider also the information related to the GMM4SA metamegamodel (see Section 4.3).

A recent AM3 extension (i.e. am3navigability [86]) allows to navigate among models both at the model and model-element level. This is fundamental in the software architecture field; a typical case is that of intra-view and inter-view navigation within an architecture description. am3navigability combines weaving models and megamodeling: weaving models establishing the links among various MDE artifacts are also part of the megamodel; then the am3navigability underlying engine provides navigability across MDE artifacts by automatically navigating the weaving models. The am3navigability extension provides the navigation links to the software architect by means of ad-hoc Eclipse views; they are shown in Figure 4.3.d. More specifically, it provides two views: (i) the Model Level view which presents all the models linked to the currently selected model (in Figure 4.3 sidModel is the currently selected model, and tldModel is linked to it), and (ii) the Model Element Level view which lists the various links at the model element level. Please refer to [86] for further information on this AM3 extension. I re-use the am3navigability engine since architecture correspondences have been defined as weaving models, so the MEGAF prototype provides inter- and intra-view navigability by automatically navigating the various correspondences defined among architecture models.

As already said, since in this work the conceptual model for architecture description of
ISO/IEC/IEEE 42010 has been defined as a metamegamodel, checking if a megamodel is structurally correct with respect to the ISO/IEC/IEEE 42010 standard consists in checking the conformity of a megamodel to the GMM4SA metamegamodel. Thanks to the generality of the EMF framework, this check is automatic in MEGAF. For example, if the megamodel contains a concern that is not framed by any viewpoint, an error message is shown in MEGAF; this is because in GMM4SA the reference between SystemConcern and ArchitectureViewpoint has cardinality “1-*” (see Figure 3.4). Of course this is not enough, so MEGAF contains a set of OCL constraints to establish properties that go beyond structure; this aspect of MEGAF is described in the next section.

EXTENSION FOR ARCHITECTURAL CONCEPTS

In order to gear MEGAF with software architecture-specific facilities, I needed to extend the AM3 core engine. In this section I describe all the features I added to the AM3 core engine by extending it programmatically.

First of all, due to its generality, the EMF editor is very basic; the AM3 core plugin provides a megamodel navigator in which the megamodel itself may be edited. However, it had to be extended it for specifying attributes and references related to GMM4SA only. So, I provide specific forms for specifying concerns, stakeholders, viewpoints, and other architecture-specific elements. Moreover, I extended also the element editors of AM3 so that they provide the means to manage, update and link the elements conforming to the GMM4SA metamegamodel. For example, MEGAF provides an editor for architecture viewpoints in which the software architect can specify which system concerns are framed by the current viewpoint; or for each architecture description element, the software architect can specify which architecture decisions affect it, and so on. Figure 4.3.c represents the extended editor for architecture models. As can be seen, an architect can easily navigate and specify relations that the current architecture model has with: (i) its model kind, (ii) the architecture views it belongs to, (iii) the set architecture decisions affecting it.

As conceptually highlighted in Section 3.1, I defined a set of OCL constraints to help software architects in checking various properties of the current architecture description. These OCL constraints are implemented as ATL queries since the whole MEGAF tooling set is based on the AMMA platform (ATL is part of it as well) and ATL provides a stable and intuitive implementation of the OCL language; further on, ATL queries can be launched either programmatically or via Ant scripts. This results in a homogeneous framework to orchestrate, manage and configure such queries within MEGAF. Results of such queries can be either stored in an external file, serialized into a specific model or simply printed to the Eclipse console. Considering the example presented in Section 3.1.2, Figure 4.3.e illustrates the Eclipse console showing the result of the NodeCheck correspondence rule execution.

The MEGAF tooling set can be extended: an extension point is provided to define how to
import specific kinds of models into a megamodel conforming to GMM4SA. For instance, a set of stakeholders and concerns may be extracted from a stakeholder diagram, or a set of decisions and rationales may be extracted from some other architecture knowledge diagram, a specific model may be extracted from an Excel sheet, and so on. In the SBSCS running example, I implemented an extension of MEGAF that is able to analyze a financial model in Excel and produce its corresponding EMF model; this allows architects to specify financial properties using Excel and consider this information in the megamodel of the SBSCS architecture description.

4.4 REALIZATION OF DULLY

In this section I will give some more details about the realization of DULLY. More specifically, Section 4.4.1 presents the concepts that are part of the $A_0$ pivot language, Section 4.4.2 presents how ADLs can be woven to $A_0$, and Section 4.4.3 presents how this weaving can be used to automatically generate model transformations between ADLs and $A_0$.

4.4.1 REALIZING THE $A_0$ PIVOT LANGUAGE

The main purpose of $A_0$ is to provide a centralized set of architectural elements with respect to which relations must be defined. The $A_0$ pivot language has been defined as the metamodel shown in Figure 4.4). In the following I describe its main metaclasses.

Figure 4.4: $A_0$ metamodel

**Architecture**: a collection of components and connectors instantiated in a configuration. It may contain also a TypeSpecification defining a set of architectural types, a
Behavior that specifies a high-level description of dynamic aspects of the system and SAinterfaces representing points of interaction between the external environment and the architecture being modeled.

**SoftwareArchitecture**: a specialization of Architecture representing exclusively the software part of the system being designed.

**SAcomponent**: a unit of computation with internal state and well-defined interface. A component can contain a sub-architecture consisting of lower-level components, connectors and their configuration. It can contain a behavioral model. Components interact with other architectural elements either directly or through SAinterfaces.

**SAconnector**: represents a software connector containing communication and coordination facilities. A connector can contain a behavioral description, a set of interfaces defining its points of interaction with other architectural elements and a sub-architecture in the same way of SAcomponents.

**SAinterface**: specifies the interaction point between an SAcomponent or an SAconnector and its environment. It is semantically close to both UML concepts of interface and port, it may contain a set of Properties and it can play the role of a link end, i.e., architectural channels can be attached to it, further it can have either input, output or input/output direction.

**SArelationship**: its purpose is that of delineating general relations between \( A_0 \) architectural elements; it can be either bidirectional or unidirectional. It is connected to any ConnectableElement, making the configuration of \( A_0 \) models as generic as possible.

**SAchannel**: a specialization of an SArelationship representing a generic communication mean; it supports both unidirectional and bi-directional communication, both information and control can be exchanged. Because inconsistencies in the design too often cause design flaws, the SAchannel is aptly left general enough so that further specification is categorically needed at a later design time.

**SAbinding**: relates an SAinterface of a component to an SAinterface of one of its inner components. I do not define intentionally the nature of the relationship between the two SAinterfaces, it would be interpreted as an equivalence relationship (i.e., the two interfaces are precisely the same), a message passing or a service call. Semantics can be either inferred from the direction of the SAinterfaces or defined through future extensions of the metamodel. SAbinding is semantically very similar to the UML Delegation Connector.

**SAtype**: type specification can be as important as any kind of topological or architectural specification; SAtypes define architectural types, so any architectural element can potentially be an instance of a particular SAtype. Each SAtype can contain a set of properties and the behavior of its instances, its internal structure is not specified. SAtype is a specialization of Type.

**SAstructuredType**: each SAstructuredType is a specialized SAtype that can con-
tain also a definition of its sub-architecture, i.e., a set of architectural elements defining its internal structure in terms of components, connectors and their interconnections. Internal elements are interpreted as default elements contained into every instance of the corresponding SAsstructuredType.

**Behavior:** represents the behavior of a component, a connector, an interface or a system architecture. It is an abstract metaclass that plays the role of a “stub” for possible extensions of the metamodel representing dynamic aspects of the system (e.g., finite state processes, labeled transition systems, state diagram, UML-like sequence diagrams).

**Development:** represents the direct relation between the architectural and the technological aspects, such as the process that will be used to develop the system, the distribution of tasks within the developing teams, the programming languages used to develop a certain component, etc. Since the Development metaclass is aggregated to Element, development details can be associated to every element of the $A_0$ model. Development is an abstract metaclass and its realization is left to future extensions of the $A_0$ metamodel.

**Business:** an abstract metaclass to be specialized via future extensions of $A_0$ and represents the link to business contexts within which software systems and development organizations exist. Its possible realizations may include a system marketing strategy, costs, product-line issues (e.g., variants and options) or generic issues regarding how the developing organization is related to the system. In the same way as Development, it is aggregated to Element in order to provide business information to every element of $A_0$ models.

**Type:** its obvious purpose is that of specifying any aspect concerning a data type definition. Types can be associated to any other architectural element of $A_0$. This is particularly useful for defining semantics within specific contexts. Types can be related through a “supertype-subtype” relationship in order to give the possibility to create hierarchical type systems within the DUALLY framework.

**TypesSpecification:** type specification diagrams contain the specification of types to correctly and orderly put them in relation or simply to keep them on an architecturally separate level. Defining a TypesSpecification element within a software architecture means that such architecture may include instances of the types defined in the TypesSpecification.

**Property:** a generic feature of a PropertyHolder, i.e., every element that is allowed to contain properties. Each Property is defined by a name, a type (PropertyType) and a value (PropertyValue). Since the Property metaclass is aggregated to Element, properties can be associated to every element of the $A_0$ model.

**PropertyType:** an enumeration containing (i) common primitive datatypes, (ii) a “reference” type that represents a reference to another element of the model and (iii) a generic type called “any” inspired by the OclAny type in the OCL language, which is considered
the supertype of every PropertyType\textsuperscript{10}.

**PropertyValue**: a metaclass defining the value of a Property. For each PropertyType it contains an attribute conforming to the equivalent primitive type. Therefore, each PropertyValue is bound to its corresponding real value; this overcomes one of the weaknesses of Acme in which the value of a property consists only in a string that exposes models to ambiguity and expressivity problems. Moreover, thanks to the extensible nature of the $A_0$ metamodel, the PropertyValue metaclass can be extended in order to contain OCL expressions providing more formal (and analyzable) means to define architectural properties.

**Group**: a logical grouping of any element of $A_0$, and it can contain architectural elements, properties, other groups and so on. The semantics of group within the $A_0$ metamodel has been strongly inspired by the xArch concept of group. It differs in the relationship with the group members only. xArch defines a group as a containment, while in $A_0$ I specify it as a plain association relationship. I believe that this provides more generality to the modeling framework since $A_0$ groups can be considered as pure logical groupings, avoiding the constraint that each member of a group must be contained into it. This also avoids the problem that would arise if a model element belongs to more than one group.

During the development of the $A_0$ metamodel, the issue of future extensions has been taken into account. I intentionally defined many minor abstract metaclasses with very generic semantics. The rationale behind this choice is that of providing different granularity of place-holders that can be extended by metaclasses of future extensions.

**Structural abstract metaclasses**

**Element**: the root of the metamodel, i.e., every metaclass specializes Element either directly or indirectly. Every Element has a name and a description field.

**TypedElement**: represents every modeling element that may have a type; so it has a reference to the Type metaclass in order to relate the TypedElement to the Type it is instance of.

**PropertyHolderElement**: represents every $A_0$ model class that can be associated with a set of properties.

**ConnectableElement**: represents all the elements that can play the role of link ends; they are the elements to which a link can be attached to.

**ArchitecturalElement**: the superclass for all $A_0$ elements that can be contained into a software architecture.

\textsuperscript{10}OCL specification: http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL.
4.4 Realization of DUALLY

Component: defines computational elements and data store elements offering services through a predefined set of InteractionPoints. Components communicate with other system elements via Links.

InteractionPoint: is contained by Components and defines a generic point of interaction between a system element and its environment.

Link: an abstract connection between model elements; semantics can be added to it through specialization, e.g., to represent communication channels, service requests, and control/data flow.

$A_0$ is designed to be a generic, expressive metamodel in order to ease the mapping of ADL-specific concepts to $A_0$ elements; the well-formedness of the $A_0$ metamodel is ensured by a set of constraints defined in OCL. For example, constraints ensure that SAbindings relate SInterfaces with same direction, that an SChannel cannot have a sub-architecture, and so on. For the sake of readability I do not describe such constraints in this dissertation. Additional constraints can be defined straightforwardly exploiting the extensible structure of the metamodel.

4.4.2 WEAVING ADLS TO $A_0$

The scope of this section is about the definition and management of weaving models within DUALLY. In the remainder of this section, Section 4.4.2 presents how DUALLY extends the graphical editor of AMW (mainly by adding the buttons to launch the HOTs and to overcome some presentation issues) and Section 4.4.2 describes the DUALLY weaving metamodel.

DUALLY WEAVING MODELS EDITOR

The weaving models editor is the graphical front-end of DUALLY. It manages the interaction between architects and the model transformation engine. It is composed of three main panels: weaving panel, left woven meta-model panel and right woven meta-model panel.

In Figure 4.5 I illustrate how the weaving model and woven metamodels are represented within the DUALLY’s editor. The weaving panel (part b in Figure 4.5) is a tree-based editor to create a weaving model, i.e., the mappings between elements of two metamodels (left and right). It is built on the base weaving panel provided by AMW, taking advantage of the reflective EMF capabilities; this avoided the effort of writing a specific editor for DUALLY and enabled the tool to rely on a tested editor also commonly used in research.

The elements of the editor reflect those of the weaving metamodel (later described in Section 4.4.2); each element is featured with its own icon and is created through a con-
Figure 4.5: Graphical interface of DUALLY

textual menu. These choices speed up and simplify the development of weaving models: if the user clicks on a given element, the contextual menu shows only the child and sibling elements that may be added to such element, in respect to the rules imposed by the metamodel.

The basic toolbar of the weaving panel of AMW was extended with two buttons (see part e in Figure 4.5) that allow the user to execute the HOT directly from the weaving panel; DUALLY automatically retrieves the information needed to launch the transformations.

The left and right woven metamodel panels (parts a and c in Figure 4.5) graphically represent left and right metamodels, respectively. These panels extend the reflective editor provided by EMF with the aim of representing the metamodels as hierarchical trees. The extension solves a problem presenting itself when dealing with hierarchical metamodels: supposing that the user is creating a weaving model between two metamodels, namely leftMM and rightMM, and that a correspondence between the Port element of leftPMM and the Interface element of rightMM must be created. At this point the architect has to specify also the binding between structural features, say the “name” attributes at both sides. If DUALLY uses the standard EMF tree editor, the Port and Interface elements contain only their own features (leaving out the inherited ones, e.g., “name”), so the architect has to navigate the tree searching the feature “name” in the parent elements of Port and Interface. This makes the creation of weaving links complex and error-prone. I extended the EMF tree editor so that each element displays all its structural features (i.e., both inherited and local).
4.4 Realization of DUALLY

The decision to render metamodels as trees was taken according to three main reasons:

- ADL metamodels usually contain a large number of elements (e.g., the AADL metamodel contains more than 250 metaclasses) and a graphical viewer could make DUALLY too complex to use;
- metamodels usually contain many hierarchical relations and the tree-view best shows this feature;
- to render models as trees is a common modeling practice, hence requiring less training from DUALLY users.

Woven metamodel panels have a toolbar (part d in Figure 4.5) that allows architects to render the ADL metamodel as an Ecore diagram, taking advantage of EMF plugins.

As mentioned earlier, the metamodels to weave form an A0-centered star topology, therefore either left or right woven metamodel panel should contain A0.

Particular attention has been paid to the management of UML profiles. The procedure that DUALLY automatically applies while loading a UML profile is composed of two main steps:

1. transform the profile into an Ecore metamodel. In so doing, I avoided having to develop specific HOTs from Ecore metamodels to UML profiles and vice versa. More precisely, I develop a single type of HOT taking as input Ecore metamodels and the corresponding weaving model.

2. add the UML metamodel package to the profile. This step is unavoidable because the XMI file of a UML2 profile contains only the extensions defined by the profile. Conversely, a profile is by definition an extension of the UML metamodel, and so it must contain both its elements and the elements of the UML metamodel. DUALLY automatically adds the UML metamodel to the profile to fill this gap. Please, note that the UML metamodel is not hard-coded in DUALLY, but rather it is dynamically retrieved from the UML2 Eclipse plugin. Therefore, DUALLY is auto-upgrading with respect to future UML versions. More details on how a UML profile can be automatically transformed into an Ecore metamodel are provided in Section 4.2.

In so doing, DUALLY becomes an integrated environment to map ADL constructs, to graphically develop the corresponding metamodels (profiles), and to perform model transformations of all architecture models conforming to the involved ADL metamodels. Furthermore, this environment is open to other modeling tools thanks to the importing mechanisms provided by the EMF Eclipse platform.
DUALLY WEAVING METAMODEL

The AMW Eclipse plugin is based on a core weaving metamodel, defining the abstract notions of weaving links (what I earlier referred to as metamodel to metamodel links) between model elements.

The AMW weaving metamodel is abstract and is thought to be as generic as possible. However, it provides an extension mechanism to define domain-specific types of link. I developed a dedicated weaving metamodel for DUALLY by extending the AMW basic one. Such an extended metamodel adds ADL-specific constructs (e.g., specific links to map structural features) and is oriented to the automatic generation of ATL transformations (e.g., there is the distinction between source and target woven elements).

Figure 4.6: DUALLY weaving metamodel

Figure 4.6 illustrates the DUALLY weaving metamodel through its Ecore diagram (a graphical formalism to define Ecore metamodels). From an higher-level point of view, it describes the types of link that a software architect can establish between the ADL metamodel and $A_0$. In the following I describe the semantics of each concept within the DUALLY weaving metamodel.

- Dually: extends WModel of AMW and is the root element of each weaving model. It is composed of the references to left and right metamodels and a set of correspondences.
4.4 Realization of DUALL Y

- **Correspondence:** represents a generic mapping between elements of the woven metamodels. It extends the WLink element of AMW augmenting it with the isAbstract boolean property. Such a property is true if the current correspondence is abstract (and should be implemented by some other correspondence), false otherwise. Correspondences may also be organized hierarchically; this is done by specifying for each correspondence a possible superCorrespondence, that is, the correspondence specialized by the current correspondence (it inherits all the feature equivalences of its super correspondence). A Correspondence can be either a DirectedCorrespondence, which is explained in next point, or an EquivalenceCorrespondence, i.e., a bidirectional correspondence.

- **DirectedCorrespondence:** it is a special kind of correspondence represents a directed generic mapping between elements of the woven metamodels. It extends the Correspondence metaclass with the condition property. Condition specifies the ATL guard of the matched rule to generate and is automatically injected into the generated ATL transformation. A DirectedCorrespondence can be of two types: Left2RightCorrespondence, or Right2LeftCorrespondence. This is a peculiarity of DUALL Y: it adds navigability semantics to weaving links. This provides many advantages: (i) easier automatic generation of different ATL transformations from the same weaving model, (ii) creation of more structured weaving models, (iii) strict relation between each weaving mapping and the corresponding generated ATL rule.

- **Left2RightCorrespondence:** this element represents a unidirectional correspondence, specifically a correspondence from an element of the left metamodel to one or many elements of the right metamodel. It contains a reference to one and only one element of the left metamodel and a reference to one or many elements of the right metamodel.

- **Right2LeftCorrespondence:** this is the opposite version of the previous element. So it contains a reference to one and only one element of the right metamodel and a reference to one or many elements of the left metamodel.

- **EquivalenceCorrespondence:** it represents a correspondence with bidirectional navigability. The woven elements of this correspondence are exactly two (one from the left metamodel and one from the right metamodel), they are equivalent from the weaving point of view. This element contains a set of FeatureEquivalences that represents the bindings between the structural features (i.e., attributes and references) of the woven elements. It contains also a boolean attribute called MatchAllFeatures, it will be used while generating model transformations and will be explained in Section 4.4.3.

- **WovenElement:** it extends AMW WLinkEnd and it is the abstract element that indicates the extremity of a correspondence. The feature variableName represents the name of the variable assigned to the element in the generated ATL transformation. The boolean feature global is true if the corresponding element must be referenced by other correspondences within the weaving model, false otherwise. Similarly to Correspondence there are three types of WovenElement: SourceElement, TargetElement, and EquivalenceElement.

- **SourceElement:** it is the source element of a directed correspondence. For example
a \textit{Left2RightCorrespondence} must contain a \texttt{SourceElement} belonging to the \textit{left} metamodel.

- \texttt{TargetElement}: it is the target element of a directed correspondence. It contains the \texttt{MatchAllFeatures} attribute and a set of \texttt{FeatureEquivalences}. Also, \texttt{constantBindings} may be associated to a target element; this entity is important when a constant value must be always associated to some feature of the target element.

- \texttt{EquivalenceElement}: it represents an element of an \texttt{EquivalenceCorrespondence}.

- \texttt{FeatureEquivalence}: it extends \texttt{AMW WLink} and defines a mapping between two features.

- \texttt{Feature}: it extends \texttt{AMW WLinkEnd} and represents a structural feature of a woven element.

- \texttt{WovenElementLink}: it extends \texttt{AMW WLink} and specifies a correspondence between a \texttt{WovenElement} (or one of its features) and a feature of another woven element.

- \texttt{Annotation}: it specifies that the target model of the generated transformation will be annotated with some key-value pairs. This entity is important when additional information must be associated to the target element. Each annotation is represented as a key-value pair, in which \textit{key} is the label of the annotation to which the corresponding annotation value will be bound, and \textit{value} is what will be annotated into the model element. Annotations may be associated to either \texttt{TargetElements} or \texttt{EquivalenceElements}.

- \texttt{ConstantBinding}: represents a mapping between a structural feature (\textit{feat} in the weaving metamodel) and a specific, user-defined constant value (\textit{const} in the weaving metamodel).

- \texttt{Constant}: represents a constant value. The string representation of the actual value of the constant is represented in the \textit{value} feature; it will be automatically injected into the generated transformations.

### 4.4.3 Automatic Transformation among Architecture Models

A weaving model between an ADL metamodel and $A_0$ specifies which elements of the ADL metamodel correspond to other elements in the $A_0$ metamodel. Those links are descriptive, that is they cannot be "executed" as they are. \textsc{Dually} provides two higher-order transformations (HOTs) that act as a bridge between the metamodeling and modeling levels, and generate ATL model transformations corresponding to the links defined in the weaving model. Figure 4.7 illustrates the configuration of \textsc{Dually} for generating ATL transformations between an ADL metamodel and $A_0$. 
4.4 Realization of DUALLY

The input of a HOT is composed of three models: (i) the weaving model, (ii) the left metamodel and (iii) the right metamodel (referring to Figure 4.7, \(\text{wm, ADL metamodel, and } A_0 \text{ metamodel, respectively})\). The output is an ATL transformation generated on the basis of the mappings defined in the weaving model (referring to Figure 4.7, it may be either \(\text{ADL}_2A_0\) or \(A_0\_2\text{ADL}\)); at a high level of abstraction, the current version of DUALLY translates each \textit{Correspondence} into an ATL matched rule and each FeatureEquivalence into an ATL binding. In DUALLY there are two HOTs: \textit{Left2Right} and \textit{Right2Left}. The former generates a transformation from models conforming to the left metamodel to models conforming to the right metamodel while the latter generates the inverse transformation. Listing 4.1 shows a fragment of the \textit{Left2Right} higher-order transformation.

```
1module DUALLyLeft2Right;
2create OUT : ATL from IN : AMW, left : ECORE, right : ECORE;
3...
4rule equivalenceCorrespondence -
5from
6  amw : AMW! EquivalenceCorrespondence
7 to
8  atl : ATL! MatchedRule {
9    name <- if(amw.name.oclIsUndefined () or amw.name = '') then
10       'rule' + thisModule.countGeneratedRules.toString ()
11      else
12        amw.name
13      endif,
14     isAbstract <- amw.isAbstract,
15     superRule <- if( (not amw.superCorrespondence.oclIsUndefined ()) ) then
16       if( (not amw.superCorrespondence.oclIsTypeOf (AMW! Right2LeftCorrespondence )) ) then
17         amw.superCorrespondence
18       else
19         OclUndefined
20     endif
21     else
```
Listing 4.1: Fragment of higher-order transformation in **DUALLY**

The header of the transformation in the listing specifies that it takes as input a weaving model (called **IN** in the listing) and two metamodel conforming to Ecore (called **left** and **right** in the listing, respectively), and generates the **OUT** model transformation. The transformation rule of this fragment is the one in charge of considering every equivalence correspondence in the weaving model in input, and producing as output an ATL matched rule with respect to the various features set in the correspondence (like `isAbstract`, and `superCorrespondence`). Bindings can also be generated implicitly by the `matchAllFea-`
4.4 Realization of DUALLY

tures attribute in a TargetElement. If this attribute evaluates to true, the higher-order transformation calculates the union of the structural features of the woven metaclasses, determines the subset of features with the same name, and automatically generates the ATL bindings among the features of the subset. This avoids the manual definition of numerous weaving links when all the features of two elements must be bound. If the user sets matchAllFeatures to true and defines other feature equivalences, these will override the corresponding implicit ones, making the editor flexible. For example, let me suppose the existence of elA and elB, in the left and right metamodels, respectively. elA and elB contain ten attributes with the same name. Figure 4.8 illustrates two versions of a weaving model that binds all the features of the elements. The weaving model (a) contains a feature equivalence for each feature to map and (b) implicitly binds such features. The weaving model (a) has complete control on the bindings, but the weaving model (b) is more readable and easy to define.

DUALLY automatically generates ATL transformations at the model level; in this dissertation I call them Basic Transformations (BT). These transformations are represented in Figure 4.7 by the arrow between mADL and mA0 and vice versa. The main difference between HOTs and BTs is that the former ones operate at the metamodeling level while the latter ones operate at the modeling level. Therefore, a HOT is usually executed only once for each ADL metamodel during the generation phase; on the contrary BTs are executed on each architecture model conforming to the ADL metamodel. An example of basic model transformation generated by DUALLY is presented in Listing 4.2.

```java
1 module DarwinFSP"A0:
2 create OUT : DarwinFSP from IN : A0;
3 . . .
4 . . . GENERATED
5 rule SAComponent_ComponentInstance { from
6   rightSide : A0!"A0::core::SAComponent"
7 to
8   leftSide : DarwinFSP!"DarwinFSP::Darwin::ComponentInstance" ( identifier <- rightSide.name,
9     declaration <- rightSide.type.first ()
10   )
11 . . . GENERATED
12 . . .
13 rule portal_SAInterface –
14 from
15   rightSide : A0!"A0::core::SAInterface"
16 to
17   leftSide : DarwinFSP!"DarwinFSP::Darwin::Portal" ( identifier <- rightSide.name,
18     type <- if rightSide.direction = #inputOutput then
19       #undefined
20     else
21       if rightSide.direction = #output then
22         #require
23       else
24         OclUndefined
25     endif
26     endif,
27     interface <- rightSide.type.first ()
28   )
29 . . .
30 . . .
31 . . .
32 . . .
33 . . .
```

Listing 4.2: Fragment of basic transformation from A0 to Darwin/FSP
The header of the transformation in the listing specifies that it takes as input a model conforming to $A_0$ (called IN in the listing) and produces a model conforming to a metamodel representing Darwin/FSP concepts (it is called OUT in the listing). The transformation rules of this fragment are: `SAcomponent_ComponentInstance` that takes as input an SA component of $A_0$ and generates a Darwin component, and `portal_SAInterface` that takes as input an SA interface of $A_0$ and generates a Darwin portal (here the type of the portal is set according to the direction of the SA interface in $A_0$). The rules contain a set of self-explanatory feature bindings.

**MANAGING THE LOSS IN TRANSLATION**

Within DUALLY, one of the most important properties to preserve is the synchronization between the various ADL metamodel, i.e., changes made on a specific (generated) architecture model must be propagated back to the other models connected to $A_0$. This is one of the key issues in languages interoperability. Reasoning by example, in Section 3.2.3 I linked two different notations to $A_0$: a UML profile for component-based architectures and Darwin/FSP. In this case a Darwin/FSP specification may be automatically obtained from a UML model, allowing the architect to perform a deadlock-freedom analysis on the Darwin/FSP specification. Therefore, once the Darwin/FSP model has been refined, as required by the performed analysis, it is crucial to propagate back those changes to the UML model in order to keep the two architecture models consistent.

![Figure 4.9: Basic Mechanism to manage the loss of information between architecture models.](image)

Various approaches have been recently proposed in order to tackle this problem. In [71] the authors provide a framework to compare current model synchronization approaches, classifying them by the nature of the involved transformations (i.e., whether they are total or partial, bijective or injective and if the reverse transformations are given or not). Since the DUALLY framework is built by following the model-driven engineering paradigm, all of these approaches can be exploited depending on the assumptions made on the transformations generated from the DUALLY weaving models. For example, if designers...
prove (or assume) that the generated transformations are total and bijective, then the corresponding approach may be used. This implies that designers have to analyze the generated transformations and make assumptions on them. Obviously, this is not always possible (e.g., transformations with many manual ad-hoc refinements are hard to classify) and for those cases, a basic and generic mechanism to keep models within the DUALLL framework consistent is needed. Figure 4.9 shows how the mechanism works.

Let us suppose that an architecture model Ms conforming to its ADL metamodel MMsource has been developed; then the software architect executes the generated transformation Msourcing2Mtarget producing a target architecture model Mt. As it can be noticed in Figure 4.9, only some of the Ms elements have been translated into Mt (i.e., the dotted part, technically called the strip (Ms, Msourcing2Mtarget) [71]) while some other concepts have not been translated.

At this point changes would be made on the Mt model obtaining the model Mt'. Now the need to synchronize Mt' with the initial Ms model arises. Applying Mtarget2Msourcing, the software architect obtains only the dotted part of the model Ms', i.e., the part that is consistent with the part matched by the initial transformation (i.e., Msourcing2Mtarget). This part of the Ms' model does not contain the unmatched elements corresponding to the white part of Ms in the figure which represents a loss of information. This mechanism provides the means to automatically store and re-add those elements when closing the round-trip journey, thus obtaining a source architecture model synchronized with the Mt' model.

More precisely, executing the Msourcing2Mtarget transformation, the part of Ms that is not involved by the transformation is stored in the MLost model. When executing the Mtarget2Msourcing transformation on Mt', we obtain a model containing only the elements corresponding to strip(Ms, Msourcing2Mtarget). This mechanism allows architects to produce a full Ms' model that contains also the elements that were in Ms - strip(Ms, Msourcing2Mtarget). I call such set of elements lost-in-translation because they are lost during the execution of Msourcing2Mtarget.

MLost conforms to a single, generic metamodel called MMLost and represented in Figure 4.10. Each model conforming to MMLost is composed of a root element containing the URI of the model (Ms in Figure 4.9) and the transformation it has been generated from (instructed Msourcing2Mtarget in Figure 4.9). The root element is composed of a set of lost-in-translation elements that may be either (i) model elements containing a reference to the corresponding model element or (ii) structural features (i.e., attributes or references to other elements). Each LostStructuralFeatureValue contains a reference to the original model element that owns it. Ref represents a reference to the element lost during the transformation; its attribute ref contains the XMI-ID of the corresponding element. The Ref metaclass can be extended to support different identification mechanisms. For example, the name attribute of the corresponding element could be used as a reference.

\[\text{\textsuperscript{11}}\text{It is important to note that this mechanism is not intended to be the unique synchronization method in DUALLL. On the contrary, this method represents the default solution to be used when other approaches cannot be applied since no assumptions can be made on the generated transformations.}\]
key if XMI-IDs are not preserved during the execution of the transformations. Within the development of the **DUALLY** framework I extended the ATL transformation engine in order to preserve the XMI-IDs while executing the transformations. Then, the XMI-ID attribute is enough to uniquely identify model elements. This provides also a good level of scalability since software architects do not need to trace models while round-tripping among architecture models. The correspondence between model elements is identified by directly referring to their XMI-ID attribute.

From an implementation point of view, I extended the higher-order transformations of **DUALLY** so that:

- the generated transformation is “instructed” so that it returns as output a target model and an additional model containing the lost-in-translation elements;
- the generated transformation takes as input a source model and a previously created lost-in-translation model and re-adds its elements to the target model.

In the current prototype of **DUALLY**, when executing a higher-order transformation a dialog appears asking the user if the transformation that will be generated must be instructed or not. In this way the user is allowed to produce three kinds of transformations:

1. **Not instructed**: the model transformation does not take into consideration the lost-in-translation mechanism of **DUALLY**;
2. **Instructed Msource2Mtarget**: the transformation creates the additional lost-in-translation model;
3. **Instructed Mtarget2Msource**: the transformation takes as input the additional lost-in-translation model and adds its elements to the target model.
In the following I describe how instructed transformations manage lost-in-translation models within DUALLY. Both Msource2Mtarget and Mtarget2Ms source are instructed by adding an endpoint rule, i.e., a rule that is executed at the end of the execution of the model transformation. This gives the possibility to check also elements that belong to the target model and make assumptions on them. Msource2Mtarget, the transformation producing also a lost-in-translation model, is instructed so that it generates a LostModelElement for each model element that is not matched by any of its rules and sets the corresponding Ref to the XMI-ID of the current not-matched element. Moreover, it creates a LostStructuralFeature for each attribute or reference that is not matched and sets a Ref element containing the XMI-ID of the model entity that owns that feature. Msource2Mtarget, the transformation that takes as input the auxiliary lost-in-translation model, performs a three-steps task:

1. for each LostModelElement it checks if the parent entity of the lost element in Ms still exists in the target model Ms’. If this is the case a fresh new model element is created looking up the element pointed by the corresponding Ref, otherwise the current LostModelElement is ignored;

2. for each LostStructuralFeature the corresponding attribute or reference is set by copying the value pointed by the corresponding Ref element. If the owner of this structural feature does not exist in the target model, the current LostStructuralFeature is ignored;

3. for each LostModelElement successfully re-applied in step 1, Msource2Mtarget restores all the values of attributes and references that do not have a default value in the original source model Ms.

4.5 **REALIZATION OF BYADL**

Similarly to MEGAF and DUALLY, BYADL relies on the Atlas Model Management Architecture (AMMA) [11]. We chose AMMA since it best fits to our technical needs, like metamodel independence, high flexibility of the composition behavior, integration to Eclipse and its modeling facilities [8] [80]. The technology we use for metamodel composition is the Atlas Model Weaver (AMW) [39], whereas model transformations are specified using the Atlas Transformation Language (ATL) [84]. Both models and metamodels are expressed via Ecore, the metamodeling language of EMF, and serialized into XMI files.

In this section I will give some more details about the realization of the various parts of BYADL. More specifically, Section 4.5.1 presents a detailed description of the BYADL composition operators and how they have been formalized. Section 4.5.2 presents how BYADL allows architects to automatically generate different concrete syntaxes to visualize and edit architecture models. Section 3.3.3 describes how semantics is provided to the
Chapter 4. Realization of the solution

ADL being extended/customized in BYADL. Finally, Section 4.5.4 describes how existing tools for the original ADL can still be used to manipulate and analyze models which have been created by means of the composed ADL.

4.5.1 Realization of the BYADL composition mechanism

In BYADL metamodel composition is the means by which the software architect either extends or customizes an architecture description language. This task is performed by applying specific composition operators to metamodels stored in the MEGAF repository. The BYADL operators, namely Match, Inherit, Reference, and Expand, have been defined building on those presented in [38, 141]. In particular, I made the operators proposed in [38] applicable on Ecore metamodels and I refined those introduced in [141] in order to lighten the composition process. For example, the deletion of elements is implicitly performed in BYADL by a slicing algorithm. The BYADL approach is somehow parallel to that proposed by Ledeczi et al. [95]: in that approach the structure of the metamodels to compose cannot change (so that original models are still valid). BYADL’s migrators permit to relax this constraint, providing more flexibility while designing and applying the composition operators. Aside from the composition operators, the BYADL conflict resolution mechanism is based on the work presented in [124].

The BYADL composition operators are defined in terms of the weaving metamodel reported in Figure 4.11. This metamodel is given in KM3, the domain specific language for specifying metamodels in the AMMA platform. The Composition metaclass represents the root of each composition model. It references the metamodels to compose (i.e., leftMM and rightMM) and contains a set of operator applications (op in Figure 4.11) which can be given in any order without changing the result. The mergeNamespaces attribute specifies if the composition will preserve the packages structure of the metamodels to compose or if the composed metamodel will contain a single root package.

![Figure 4.11: Definition of BYADL metamodel composition operators](image)

The default behaviour of the composition is to consider the union of all the metaclasses in both leftMM and rightMM and to interrelate them by applying the specific operators. Such behavior may vary depending on the sliceLeft or sliceRight attributes. If sliceLeft is true, then only the portion of the left metamodel involved in some operator application
will be part of the composed metamodel. The same mechanism holds for sliceRight. The relevant portion of metamodel is computed by executing a slicing procedure before calculating the union of the metaclasses in leftMM and rightMM. This slicing procedure is based on a slicing algorithm for UML models [5]. Let MM be a metamodel and let SC be a subset of the elements in MM; slice is defined as follows:

\[
\text{slice}(SC) = SC \cup \bigcup_{c \in SC} \text{slice}(\text{neighbour}(c))
\]

where neighbour(e) is the set of all superclasses of c, of all classes referred (both with association and aggregation) by c, and of all types of attributes in c. It is important to note that even though slice is defined as a set of classes, since each class contains also references to other classes, the final result is a part of the metamodel MM with both classes and their relationships. This facility is useful while composing a large metamodel and just a part of it is relevant for composition.

In PreferenceElement, the optional preferred attribute is used to solve conflicts during composition: if the composition engine has to make an arbitrary choice, then it chooses the option specified by the preferred element. If preferred is defined both in Composition and in an operator, then the operator’s preferred “overrides” the one of Composition. Each operator is always applied on two metaclasses that I refer to as source (s) and target (t) in the remainder of this section. The composition operators of BYADL are:

- **Match.** s and t semantically overlap and then they are merged into a single metaclass c in composedMM. c contains the union of all the structural features (i.e., both attributes and references) of s and t. The features referenced by the ignoring aggregation are discarded. The merging reference specifies if two features must be merged before calculating the union of the features of s and t. This is necessary in case two features are semantically the same, but they are syntactically different. The supertypes and subtypes references are merged. The className optional attribute represents the name of the merged metaclass. If this attribute is not specified, then the preferred metaclass’s name is chosen, otherwise the name will be the concatenation of the names of s and t (the latter case is signalled to the user with a warning). The namespace attribute specifies the package that will contain the c metaclass. If it is not set, then the root package of composedMM is chosen. This operator is subject to semantic checks (and corresponding solving mechanisms) that handle possible conflicts between properties of s and t. For example, if both metaclasses are abstract then the resulting metaclass c is abstract, otherwise it is not abstract. The same mechanism holds for the interface property.

- **Inherit.** This operator specifies that s will be a subtype of t in the resulting composed metamodel. If its application results in a cycle in the inheritance tree, then it is not executed and a warning is raised. The abstractTarget attribute specifies if the t metaclass will be abstract (then it cannot be instantiated) in the composed metamodel. If the exclusive attribute evaluates to true, then the metaclass corresponding
to $s$ in composedMM will extend only $t$, discarding all its previous inheritances in its original metamodel.

- **Reference.** In the composed metamodel $s$ references $t$. All features of the new reference (e.g., lowerbound, name, opposite) can be set while applying this operator. If the new reference $ref$ overlaps an existing one, then it replaces the old one. The $\text{oppositeRef}$ feature is used to set the opposite of $ref$ to the result of another application of the $\text{Reference}$ operator.

- **Expand.** The attributes of $s$ (leaving out the attributes specified in the $\text{ignoring}$ aggregation) are copied into $t$. The standard merge operation is executed to manage attributes with the same name. If the $\text{deep}$ attribute is true, then all attributes of $s$ (including the inherited ones) are recursively copied into $t$. The $\text{sliceSource}$ attribute specifies if the $s$ metaclass is still part of the composed metamodel. If this property evaluates to true, then: i) the $s$ metaclass is deleted from composedMM, ii) all the metaclasses extending $s$ will extend the direct supertypes of $s$, iii) all the references whose type is $s$ are deleted from the composed metamodel. The $\text{ignoring}$ and $\text{merging}$ aggregations have the same semantics as in the $\text{Match}$ operator.

The composition engine performs various semantic checks to avoid incidental conflicts, such as attributes overlapping with different types, inheritance cycles, and so on. The composition phase solves also discovered conflicts. The rationale that guides the solution of conflicts in BYADL is to propose a combination of operator’s preferred element, if specified, and to propose a default behavior for each operator in the remaining cases. If structural features overlap, merging and feature equivalence detection mechanisms have been implemented, inspired by the UML $\text{PackageMerge}$ algorithm [40]. Without going into the details, the main principle is to keep consistency while trying to preserve, as much as possible, the semantics of the metamodels to compose.

It is not infrequent that structure and semantics of a MOF metamodel are restricted through OCL constraints. Such constraints must be preserved since, as stated above, the composed metamodel must embrace as much semantics as possible from the initial metamodels. In that sense, the BYADL composition engine copies the OCL constraints into composedMM and refines them according to the composedMM package structure. By doing that, within the resulting composedMM, incidental ambiguities and constraint clashes are reduced to a minimum.

The BYADL composition engine reuses the user interface of AMW. LeftMM and RightMM are rendered using a tree-based editor and operators are graphically applied via a central panel. We extended the AMW interface so that the semantic links to the $A_0$ metamodel guide the application of the composition operators. More precisely, once applying an operator, BYADL proposes as target only metaclasses that are semantically compatible with the source metaclass. So, if the architect is applying an operator between two metaclasses $c_1$ and $c_2$ and $c_1$ is selected, then the tool allows to select only metaclasses semantically compatible with $c_1$ (i.e. metaclasses that have been linked to $A_0$ concepts related in some way). For example, it is possible to match $c_1$ and $c_2$ only if they are semantically linked to
the same $A_0$ metaclass or, if the $A_0$ metaclasses are different, an inheritance relationship (either direct or not) must exist between them. In this respect, semantics compatibility of metaclasses depends on the type of operator being applied.

Figure 4.12: BYADL composition mechanism

Once a weaving model specifying how the ADL metamodel and the extension metamodel fit together, it can be used to automatically compose them in order to get the target composed metamodel. As shown in Figure 4.12, the weaving model is the input of a transformation process which generates a composed metamodel according to the relations specified between the single source metamodels. More specifically, the composition model allows the composition of two metamodels, (leftMM and rightMM in figure) into a metamodel composedMM by executing the Composition2Ecore ATL transformation. This transformation takes as input leftMM, rightMM, and the Composition Model and produces composedMM by executing the logic defined for the applied operators. It is important to note that Composition2Ecore is independent from the metamodels being composed, and its inner logic is generic since the composition directives are specified in the weaving model linking the metamodels being composed.
4.5.2 Support for Concrete Syntaxes Generation in bYADL

Visualization features and editors are an important and ineluctable aspect of architecture description languages. In fact by means of editors software architects can draw, design, view, and reason about the software architecture of the system they are modeling. By using bYADL, three different kinds of editors can be adopted, each characterized by a distinguishing usability level and a certain effort to be implemented. In particular, software architects can use:

- **tree-based editor**: this graphical editor is obtained for free by EMF. Similarly to the tool support of xADL [32], this editor shows collapsible and hierarchical tree-like structures; it is very simple and useful for large size software architectures, even though it’s usability level is quite low. The left part of Figure 3.32 shows an example of this editor at work.

- **textual editor**: starting from the metamodel of the created ADL, a textual editor conforming to the Human-Usable Textual Notation (HUTN) specification\(^\text{12}\) can be automatically generated. HUTN is a generic specification that provides a concrete language for any MOF model. The produced textual editor supports also syntax highlighting and automatic conformance check with respect to the metamodel (i.e., the abstract syntax of the composed ADL). The usability level of this editor is slightly higher than the tree-based one. In fact, both require a deep knowledge of the metamodel even though in many cases a textual syntax is preferred to the tree based editor. An example of this editor is shown in the right part of Figure 3.32.

- **graphical editor**: software architects can also produce a graphical editor with customized graphical elements. This is possible by means of the EuGENia\(^\text{13}\) tool. This tool automatically generates the models needed to implement an editor from a single annotated Ecore metamodel. EuGENia is based on the Eclipse Graphical Modeling Framework (GMF)\(^\text{14}\) that provides a generative component and runtime infrastructure for developing graphical editors for EMF models. In EuGENia, suitable annotations allow developers to identify the different roles of the elements (such as the root object of a metamodel) and to specify graphical properties (such as `border.color`, `label.icon`, etc.). Let $ADL_{\text{new}}$ be the extended ADL under development, let $ADL_1$ be the ADL selected as ADL to be extended, and let $MM$ be the metamodel representing the extension. The graphical editor features allow $ADL_{\text{new}}$ to inherit graphical elements of $ADL_1$ and $MM$, under the assumption that annotated Ecore metamodels were available for $ADL_1$ and $MM$. In case of a graphical element for an element of $ADL_{\text{new}}$ exists both in $ADL_1$ and $MM$, the `preferred` field of the operator being applied for the composition can help to disambiguate. Sometimes $ADL_1$ and $MM$ provide separated views that should be kept separated.


\(^{14}\)GMF: http://www.eclipse.org/modeling/gmf.
and sometimes they should be merged. BYADL, by means of its graphical interface, allows the software architect to select the most suitable solution. For instance in Figure 3.33 it can be seen an example of graphical editor in which Darwin and an extension for modeling connectors have been integrated into a single view (white area in the figure), while BPMN has kept its separated view (gray area in the figure). Links between these two different views are added by hand in Figure 3.33 to graphically show existent relationships among elements of these views.

Furthermore, the semantic links specified among ADL1 and A0 and among the MM and A0 enable the possibility to inherit the graphical annotations (and hence the graphical representations) of the A0 linked elements. In other words, A0 substitutes ADL1 or MM if they do not have an annotated Ecore metamodel. Finally, the obtained GMF editor can be polished and optimized even though this step requires a deep knowledge of GMF.

The generation of graphical editors is in its prototypal version and currently it is not able to solve subtle conflicts that the different graphical elements may have. This point is better discussed while discussing the limitations of the approach in Section 5.5.

To summarize, there are three different possibilities in BYADL to produce an editor for the ADL being developed. Depending on the effort and the knowledge that the software architect wants to invest, tree-based, textual or graphical editors can be conceived, each with a different usability level.

4.5.3 SUPPORT FOR PROVIDING SEMANTICS TO A COMPOSED ADL IN BYADL

As anticipated in Section 3.3.3, the semantics of the composed ADL is given by means of semantic relationships between the concepts in the composed ADL metamodel and concepts in the A0 metamodel. By means of such relationships, the elements of the composed ADL are semantically linked to elements of A0.

It is important to recall that all metamodels stored in the MEGAF megamodel are semantically linked to A0. This means that software architects do not need to manually define the set of semantic links between a composed ADL metamodel MMADLx and A0. Indeed, at the time of metamodels composition, BYADL automatically generates an additional weaving model containing the semantic links between the composed ADL metamodel and A0. Figure 4.13 shows how the automatic generation of the semantic links between the ADL metamodel and A0 has been realized.

Basically, at the time of composition the involved modeling artifacts are: MMADLx is the metamodel of the ADL being extended/customized, MMext is the metamodel containing the concepts that will extend MMADLx, WMcomp is the weaving model containing the operators specifying how MMADLx and MMext are being composed. Since the involved
metamodels are stored in the MEGAF repository, they are already semantically linked to $A_0$; thus the $MM_{ADLx}$ and $MM_{ext}$ metamodels are linked to $A_0$ by means of $WM_{sem1}$ and $WM_{sem2}$ weaving models. $MM_{comp}$ is generated after the execution of the generic model transformation of BYADL described in Section 4.5.1. At this point, the only artifact that is missing is the $WM_{sem3}$ which semantically links $MM_{comp}$ to $A_0$. BYADL provides a generic model transformation called Comp2Sem which automatically generates the $WM_{sem3}$ weaving model starting from the set of the involved modeling artifacts. Fundamentally, for each concept $c$ of the composed ADL metamodel, the logic of the Comp2Sem transformation is the following:

- if $c$ is not involved in any composition operator application (i.e., if it has been simply copied from an element $c_{source}$ in one of the two metamodels being composed), the transformation checks which element $c_{A0}$ within $A_0$ is semantically linked to $c_{source}$ and then creates a weaving link in $WM_{sem3}$ between $c$ and $c_{A0}$. If no semantic link has been defined between $c_{source}$ and any element in $A_0$, then $c$ will not be linked to any element of $A_0$.

- if $c$ is involved in either an Inherit, Reference or Expand operators application, basically it has also been copied from an element in one of the two metamodels being composed. Thus, a weaving link is created in the same way as in the first case.

- if $c$ is the result of a Match operator application, then the $s$ and $t$ operands of the applied operator must be considered. By construction, $s$ and $t$ are themselves semantically linked to some element of $A_0$, I will call them $s_{source}$ and $t_{source}$. At this point, the transformation checks if in the composition weaving model $WM_{comp}$, either $MM_{ADLx}$ or $MM_{ext}$ has been identified as the “preferred” side of the composition (either locally to the match operator being applied, or to the whole weaving model). If this is the case, the transformation creates a semantic link between $c$ and the $A_0$ element previously linked to the preferred element in the composition. Otherwise, the transformation gives priority to the ADL metamodel; that is, a semantic link between $c$ and the $A_0$ element semantically linked to the operand belonging to $MM_{ADLx}$ is created.
4.5 Realization of BYADL

Once the weaving model establishing semantic links between the newly composed ADL metamodel and $A_0$ has been created, the concepts of the composed ADL metamodel inherit the semantics of $A_0$.

4.5.4 Realization of the BYADL Model Migrators

As previously said, BYADL provides software architects with the means to create customized ADLs by extending existing ones. However, in this scenario it is important that tools supporting original ADLs can still be used to manipulate and analyze architecture models which have been created by means of the new composed ADLs. In order to cope with such situations, BYADL provides two higher-order transformations which are able to generate specific model migrators. In particular, the Composed2SingleMigrator Generator reported in Figure 4.14 generates a Composed2SingleMigrator (see Figure 3.26) starting from the metamodels being composed and the weaving model specifying their composition.

![Figure 4.14: BYADL model migrators generation](image)

This generated transformation takes as input a model $Model_C$ conforming to the meta-
model composedMM and generates three different models: Model_L, Model_R, and Trace. Model_L and Model_R contain the elements in Model_C which can be represented by means of the metamodels leftMM and rightMM, respectively. The Trace model contains the information which is required to compose models by means of the generated transformation Single2ComposedMigrator described in the rest of this section.

Listing 4.3 reports a small fragment of the Composed2SingleMigrator Generator. Essentially, each metaclass in the composed metamodel is taken into account to determine how it has been obtained by the composition process. Depending on the operator which has been used, a corresponding transformation rule is generated. For instance in case of classes obtained from a match operation, the matchManagementRule() (see lines 3-4) is executed to generate a transformation rule which is able to decompose the instances of the considered matched class.

```
1... helper context ECORE! EClass def: toRuleString(): String =
2  if not self."abstract" then
3    if self.inherits() then
4      self.inheritManagementRule()
5    else
6      if self.comesFromMatch() then
7        self.matchManagementRule()
8  ...
```

Listing 4.3: Excerpt of the Composed2SingleMigrator Generator

The left part of Figure 3.32 reports a sample model conforming to the composed metamodel. The model consists of Darwin description augmented with modeling elements provided by a metamodel connMM representing software connectors. For example, the element NetworkType is a connector declaration which can be given because of a match operator between the metaclass ComponentDeclaration in Darwin and the ComponentDeclaration metaclass in connMM.

The generated rule ConnectorTOComponent reported in Listing 4.4 is able to decompose all the source connector declarations (like the element NetworkType above). It generates a component declaration in the target Darwin model, and a connector declaration in the target connMM model. Additionally, the migrator generates a trace model in order to store the elements which have to be considered together when the separated models have to be recomposed.

```
1 create OUTM1 : MM1, OUTM2 : MM2, OUTTRACE : TRACE from IN : MM12;
2... rule ConnectorTOComponent -
3  from s : MM1! ConnectorDeclaration
4    to t : MM1! ComponentDeclaration 
5      hasException <- s.hasException,
6    ...
7    t2 : MM2! ConnectorDeclaration ()
8  do -
9    thisModule.createTraceLink(t, t2, #match);
10   -
11   
```

Listing 4.4: Excerpt of the generated Composed2SingleMigrator

Being more precise, a trace model is another weaving model consisting of trace links. Each trace link relates two model elements which are instances of metaclasses that have
been composed. Figure 4.15 reports a fragment of the trace model generated by applying the \textit{Composed2SingleMigrator} on the sample model in Figure 3.32. The last trace link relates the \textit{NetworkType} component declaration contained in the left Darwin model with a connector declaration in the right \textit{connMM} model. Such model elements have been generated by applying the rule reported in Listing 4.4 on the source \textit{NetworkType} connector declaration.

As previously said, BYADL provides software architects also with the \textit{Single2Composed Migrator Generator} depicted in Figure 3.27. This generator produces a \textit{Single2Composed Migrator} able to generate a model \textit{Model}_C conforming to a composed metamodel \textit{composedMM} starting from \textit{Model}_L conforming to \textit{leftMM}, \textit{Model}_R conforming to \textit{rightMM}, and a trace model obtained during a decomposition as explained above. For instance, by applying the generated \textit{Single2Composed Migrator} to the models reported in Figure 4.15, a composed DarwinConn model is obtained. The trace links drive this operation since they maintain the elements which contribute to the generation of a same target element. For instance, a target connector declaration named \textit{NetworkType} will be generated because of the last trace link reported in Figure 4.15. Without this information the migrator is not able to distinguish the model elements which have to be composed from those which have to be simply copied to the target model.

The migrators have been conceived by inheriting the work done in [21, 24]. In general, adapting models with respect to the occurring modifications in the corresponding metamodel is a challenging task. In fact there are many kinds of metamodel changes that have to be taken into account and which can be distinguished into \textit{non-breaking}, and \textit{breaking}. The former are metamodel modifications which do not brake the conformance of the
existing models in contrast to the modifications which brake their conformance and that can be in turn classified into resolvable and unresolvable. An example of a breaking and resolvable modification is the deletion of a metaclass. In this case, existing models can be automatically adapted by eliminating all the instances of the deleted metaclass. However there are modifications (breaking and unresolvable) which require human interventions for their resolution, like the addition of an obligatory metaclass.

According to the semantics of the provided operators, and to the metamodel difference classification available in [21], in our case the migration of models can be automatically performed. This is because i) breaking and unresolvable modifications do not occur and ii) the trace models produced by any Composed2Single Migrator contain the necessary information to properly select the model elements which have to be composed by the corresponding Single2Composed Migrator.
In this chapter I will firstly present the application of the approach described in this dissertation to a real-world case study called BusOnAir\(^1\) and then in Section 5.5 I will provide a reflection about the proposed approach in light of the various issues arose during the case study and the various techniques that have been used to face them. It is important to note that the reflection provided in this chapter is also driven by the other applications of the approach that I presented throughout this dissertation, they are:

- Subscription-based Sensor Collection Service (SBSCS). MEGAF has been used for modelling an ad-hoc architecture framework for the SBSCS system and then I provided an architecture description of SBSCS by applying the defined framework. Please, refer to Section 3.1.2 for further details about the application of MEGAF on the SBSCS system.

- Adaptive Cruise Controller (ACC). The software architecture of the ACC system has been defined by using a dedicated component modeling language called SaveCCM, and then DUALLy has been used to automatically transform (and propagate changes) among the SaveCCM model, a UML model, and a finite state process specification. Please, refer to Section 3.2.3 for further details about the application of DUALLy on the ACC system.

- Integrated Environment for Communication on Ship (IECS). BYADL has been used to create a dedicated architecture description language for the IECS system with the main goal of covering the main concerns that stakeholders had on the IECS system (e.g., fault tolerance, direct link to the development process, etc.). Please, refer to Section 3.3.5 for further details about the application of BYADL on the IECS system.

In the remainder of this chapter I will focus on the application of the proposed approach to the BusOnAir system. The project is about a generic information system for managing real-time information about public transportation systems and making them available to the user as web services. More information about BusOnAir are provided in Section 5.1.

\(^{1}\)http://www.busonair.eu
Figure 5.1 shows the workflow taken in this case study. Basically, a dedicated architecture framework for BusOnAir is firstly created, and then its architecture description is defined by leveraging the features provided by MEGAF, DUALLY and BYADL.

![Workflow diagram](image)

Figure 5.1: Workflow followed in the BusOnAir case study

Clearly, there may be iterations between the various activities within the framework, however, for the sake of clarity I did not show them in the figure. Fundamentally, this case study can be divided into two main phases:

1. **Architecture Framework definition**. In this phase a dedicated architecture framework for the BusOnAir project is created in order to precisely focus on its specific peculiarities and technological aspects. Section 5.2 will focus on this part of the case study, with a dedicated section to each activity performed (e.g., definition of the needed architecture viewpoints, import of the needed ADLs metamodels, etc.). BYADL will be used to extend an ADL to better fit with our modeling needs (see Section 5.3).

2. **Architecture Description definition**. In this phase the created architecture framework is used to define an architecture description of the BusOnAir system. So, each viewpoint defined in the framework will be instantiated into an architecture view, correspondences between views are checked with respect to the correspondence rules defined in the framework, and DUALLY will be used to automatically transform among different architecture models, etc. Section 5.4 will focus on this part of the case study.
5.1 THE BUSONAIR CASE STUDY

BusOnAir is a project developed at the University of L’Aquila (Italy) and I am part of the team that created it. BusOnAir was born in 2010 and it actually started as a small project for participating to a business plan competition organized by the University of L’Aquila. The main goal of that competition was to support the diffusion of innovative skills within the academic world, to promote the creation of academic spin-offs that transform scientific ideas into business ideas, so contributing to the development and strengthening of regional leadership in the area of technological innovation. Currently, we are working on BusOnAir on two complementary sides: firstly we are completing the development of a first prototype of the system, and secondly we are evaluating the possibility to found an academic spin-off in order to create a business out of the project.

Figure 5.2 shows an overview of the various functionalities of BusOnAir. The main idea behind BusOnAir is the ability of the system to track in real-time the position of each bus within the city. When a potential traveller needs to go to a specific destination, it fires up a mobile application that allows him to calculate the nearest bus stop to go there. Also, given a specific bus stop, the mobile application allows the user to get the (actual) arrival time of all next buses stopping there.

Once the bus arrives at the bus stop, the traveller checks in by using a plastic smart-card (he uses it instead of paper tickets). Similarly to a traditional debit card, the smart-card can be topped up with credit when needed; this operation can be performed on line, through

\[2\text{http://bpc.univaq.it}\]
a dedicated website. That website provides also real-time information about bus routes and their arrival time at bus stops. Once arrived at his destination, the traveller checks out using the same smart-card he used to check in; this is the moment in which the system charges the amount of money corresponding to the trip. During the trip, the bus can get full (people on the bus is counted by the check-in device at the entrance of the bus) and, based on this information, the system may decide that a second bus is needed. By doing this, the level of the service provided to the user is kept at an acceptable level.

Our BusOnAir data center (which I call ”central station” here) manages all the data coming from the buses in the streets, and updates the information on the website and on the mobile applications. Logging and statistical analysis are also part of the tasks performed in the central station. It is important to note that the BusOnAir software in the central station is generic and independent from the the travelling agency being served. That is, our system scales well when serving a large number of travelling agencies. Indeed, when adding a new travelling agency X, the only input that we need is an XML file describing the various bus routes of X and containing some other additional parameter. The software running in our data center will manage autonomously the new travelling agency and will configure itself in accordance with the routes and parameters specified in the XML file.

On a gross grain, the BusOnAir system is composed of three main sub-systems:

- The Bus System (BS)
- The Central Station system (CS)
- The Mobile App system (MA)

The Bus System runs on each bus of the travelling company; it is composed of hardware and system devices devoted to monitoring the status of the bus. Its main sub-systems are: a GPS receiver which tracks the current location of the bus at a given moment, an RFID reader for managing the payment of the various trips of the users, a network device for communicating with the CS system, and a central sub-system to coordinate all the above mentioned sub-systems.

The Central Station system collects real-time data about the position of all the buses travelling within the city in order to provide an overall view of the current situation in the city. An additional sub-system may interact with an external service that provides real-time information about traffic on the streets, and may be used to more precisely calculate the arrival times of the various buses to their next bus stops. The CS system manages also all the payment requests generated from the RFID readings made on the Bus System. The CS system must be accessible exclusively in the central station and from defined users and roles. The CS system has to keep a secured log of all the operations; also, statistics need to be prepared periodically. The CS system contains also a software module for adding new travelling agencies to the system and to auto-configure itself accordingly to the bus routes of the travelling agencies being added to the system. The central station makes travelling information available to the mobile apps by means of a well defined REST API.
The **Mobile App** system runs on mobile devices and can perform three main tasks: (i) to top-up the card of the user with a certain amount of money, (ii) given a certain destination address, to calculate the nearest bus stop for taking the bus that will bring the traveller to the destination, and (iii) given a certain bus stops, to provide the arrival time of the various bus stopping there. The CS system provides to the MA system the needed information to perform the tasks at points (ii) and (iii) by means of its REST API.

The current prototype of BusOnAir mainly focuses on all the features described so far; the cross-cutting aspects related to the smart-card and the payment of the trips are left apart from the short-term development objectives of BusOnAir because they involve a stronger investment from travelling agencies and this may hamper the usage of BusOnAir in this initial phase.

In the remainder of this chapter, I will describe how a dedicated architecture framework for BusOnAir has been defined, and how an architecture description conforming to the newly defined framework has been created.

### 5.2 Architecture Framework

In this section I will use **MEGA** to define **BOAAF**, the BusOnAir Architecture Framework. It is specifically developed for defining the architecture description of systems which are similar to BusOnAir. To obtain this goal, there are several steps involved. The first step (see Section 5.2.1) is to identify all the stakeholders of the system, as from their perspective it is easier to reason about the importance of certain requirement. Next, in Section 5.2.2 I will describe the various viewpoints which frame the stakeholders concerns, and in Section 5.2.3 I will present the ADLs and other modeling notations which will be used to describe the system, and which viewpoints they belong to. Finally, I will present in Section 5.2.4 the correspondence rules that will ensure that the various architectural elements within the framework are consistent. It is important to note that we could also chose a different strategy for architecting BusOnAir; for example, we could use an already existing architecture framework, we could use different ADLs for architecture modeling, we could focus on other concerns, etc. In light of this, we created a dedicated framework for BusOnAir for two main reasons: (i) because this is the first time we are developing a system which falls in the GPS-enabled service-based system domain (and thus we did not have at hand suitable viewpoints for this kind of systems), and (ii) because we wanted to test the applicability of **MEGA** in a real-world scenario, in which each aspect of the approach is exercised in the context of a complex development process and in combination with other aspects of **MEGA**.

For the sake of presentation, in the context of this dissertation I will focus only on a subset of the architectural issues we faced, leading to an architecture framework which is easier to describe and to a more understandable architecture description. More specifically, in this case study I will mainly focus on the technical parts of the BusOnAir system, leaving out less technical aspects like financial issues, project management, and other
aspects related to the "business side" of the project. We are currently evaluating the possibility to make available to the public the complete BusOnAir architecture framework and architecture description, especially in light of the confidentiality of the considered information.

Figure 5.3: BusOnAir Stakeholders and Concerns

Figure 5.3 shows an overview of the resulting architecture framework developed for BusOnAir. Each part of the figure will be discussed in next sections, together with the various relationships between the architectural elements of the framework (e.g., which viewpoints frame a concern, which concerns does a stakeholder have, and so on).

5.2.1 SYSTEM STAKEHOLDERS AND CONCERNS

In this section, I identify the BusOnAir system stakeholders and their system concerns which drive the BusOnAir architecture. The following stakeholders are considered and identified:

- **Software Architect**: the person who is in charge of architecting the software part of the system; that is, the process of conceiving, defining, expressing, documenting, communicating, certifying proper implementation, maintaining and improving an architecture throughout the BusOnAir system’s life cycle [77].
5.2 Architecture Framework

- **System Engineer**: the person who deals with all the hardware aspects of the BusOnAir system; basically, in the context of the BusOnAir project, a system engineer can be considered as the "hardware counterpart" of the software architect. System engineers manage the RFID readers on the buses, the infrastructure of the central station sub-system, etc.

- **Developer**: the person in charge of implementing the software running in either the Bus, the Central Station, or the Mobile App sub-systems. Clearly, people belonging to this type of stakeholders may have very different backgrounds: they may be server-side developers, mobile app developers, web front-end developers, and so on; for the sake of simplicity, in this case study description I will treat them as a single type of stakeholder.

- **End-user** (traveller): the person who potentially will take the bus. They use the busOnAir system (through either a mobile or desktop/laptop device) to get information about which bus they have to take to get to a specific destination. They want to know also where is the nearest bus stop, the arrival time, together with other useful information. End-users will use a personal, dedicated transport debit card to check-in and check-out from buses, and they can manage card credit on-line.

The following stakeholders have been considered, but we determined to be not applicable for the simplified architecture framework described in this case study: financial manager, system administrator, bus driver, project manager, maintainer. Figure 5.4 shows all the above listed system stakeholders and puts them in relation with the concerns they hold with respect to the BusOnAir system.

![Figure 5.4: BusOnAir Stakeholders and Concerns](image)

Also in this case, the concerns shown in the figure represent a subset of the actual concerns that all the stakeholders have with respect to the BusOnAir system; here we mainly focus on the technical ones, they are:

- **Modularity**: the design of all the subsystems must deal with the intrinsic complexity of the BusOnAir system; thus, the interaction of each component of the system with all the other components must be clearly defined. The architecture of the system must be composed of self-contained units designed to be interchangeable and/or recombinable. Each component must be easy to upgrade and maintain as independently as possible from the others.
• **Distribution**: this concern is about the distribution of the various software components throughout the BusOnAir system. So, this concern is about the how, and why, the BusOnAir system is distributed.

• **Concurrency**: the central station sub-system should be able to concurrently handle bus-loads (almost 4000) of people checking-in/out at the same time. It must also be designed to support in the future different travelling agencies at the same time.

• **REST**: the central station must be agnostic with respect to the mobile applications accessing its information. For this purpose, we deliberately decided to implement its web services as a RESTful web API. This allows different mobile applications to access the information about the bus trips in an homogeneous manner.

• **Dependability**: as defined by the IFIP 10.4 Working Group on Dependable Computing and Fault Tolerance, dependability is a generic attribute to describe “the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers”. Dependability includes as special cases such attributes as reliability, availability, safety, security. Basically, dependability is a macro-concern embracing other concerns like: how are exceptional situations and faults handled? What security mechanisms are to be used? How is information managed and kept consistent? Etc.

5.2.2 **ARCHITECTURE VIEWPOINTS**

The BusOnAir system uses three architecture viewpoints:

1. System VP
2. Behaviour VP
3. WebServices VP

In the following I will describe the rationale that drove us to use the above mentioned architecture viewpoints. The **System viewpoint** is used to break down the BusOnAir system into a set of components, to show how they relate to each other, to better reason on their distribution throughout the BusOnAir system, and to identify which components are “more sensible” with respect to the dependability concern. The **Behaviour viewpoint** is used to reason on how the various components of BusOnAir can be executed concurrently (especially those in the Central Station sub-system), and to check some properties related to dependability (such as deadlock freedom of the components, and their reliability). The **WebServices viewpoint** is used to describe the RESTful web API of the Central Station sub-system.

5.2 Architecture Framework

Figure 5.5: BusOnAir Viewpoints and Model Kinds

Figure 5.5 shows the above mentioned viewpoints and visualizes which system concerns they frame (system concerns have been explained in the previous section). The upper part of Figure 5.5 shows also which model kinds we defined in the context of BusOnAir. More specifically, we distinguish between static and dynamic model kinds. A static model is a description of the structure of the components of the system, whereas a dynamic model describes the behaviour of those components at run-time. Furthermore, a dynamic model is more flexible of a static one since it can change over time as it shows what a architectural entity does with respect to the many possibilities that might arise in time. This is why the System and WebServices viewpoints are associated to the static model kinds, whereas the Behaviour viewpoint is associated to the dynamic model kind. In the next section I will describe how specific Architecture Description Languages (ADLs) define concretely those model kinds.

5.2.3 Used Architecture Description Languages

Figure 5.6 shows the Architecture Description Languages that will be used to define the architecture models of the BusOnAir system. More specifically, the architecture view governed by the System viewpoint will contain two models: the first one is a UML model for describing the various components of the BusOnAir system, the second one conforms to Diaspec, a declarative language for describing applications that orchestrate entities interacting with an environment (i.e., all the components which will be deployed on the buses). We need two different models for the System viewpoint because on the one hand Diaspec provides facilities for code generation and simulation, and on the other hand UML is a more commonly known description language, thus UML may be used for sharing architectural knowledge among the different stakeholders involved in the project. Additionally, UML is a good starting point for defining the behaviour of the system (via state machine diagrams) and for natively associating behavioural specifications to the UML components of the system.

The Behaviour view will contain two models as well: the first one is the same UML model of the System view (in the ISO/IEC/IEEE 42010 standard an architecture model can belong to one or more views) in which each component will contain also a state machine
describing its behaviour, the second one is a Finite State Process (FSP) model describing
the behaviour of the components of BusOnAir from a more formal perspective, which
allows architects to perform property checks and deadlock freedom analysis. Finally, the
WebServices view will contain a model conforming to a metamodel for specifying REST
services (RESTLANG in the figure).

In the remainder of this section I will give an overview on the above mentioned languages.
For each language I will provide the following information: (i) brief overview of the main
constructs of the language, (ii) facilities offered by the tool(s) associated to the language,
(iii) how I obtained the metamodel of the language, (iv) which tags I associated when
importing the language into MEGA, and (v) how I linked the metamodel of the language
to $A_0$.

For what concerns $A_0$, it is important to note that in this case study I preliminary ex-
tended it with behavioural concepts, so that behavioural information can be preserved
when passing from an architectural language to another. Structural information is already preserved thanks to the core elements of $A_0$ (e.g., SComponent, SConnector, etc.). In this context, I do not need to extend $A_0$ also with information related to the WebServices viewpoint since we are not planning to reuse those information outside the WebServices viewpoint. Figure 5.7 shows how $A_0$ has been extended by means of the extensibility approach defined in [37].

The proposed extension gives the possibility to associate behavioural state-based descriptions to $A_0$ architectural elements. This is done by a two-step process: (i) definition of StateDiagramMM, a basic state diagram metamodel and (ii) relating elements of StateDiagramMM to abstract metaclasses of $A_0$. For the sake of simplicity I defined a very simple and generic state diagram metamodel; its semantics is taken from [69]. Figure 5.7 shows the StateDiagramMM, which contains a main class called StateDiagram composed of states and transitions. Each state can either be generic, initial or final and may be hierarchically structured, i.e., it may contain the definition of a sub state diagram. Each transition may be labelled and a guard may check whether it can occur or not. It should be noted that I defined a simple extension metamodel since the aim of this extension is just to keep behavioural information while passing from an architecture model to another by using DUALLY. without going into the details on how the initial $A_0$ metamodel and StateDiagramMM have been related (see [37]), the resulting extended $A_0$ metamodel contains all the metaclasses from both $A_0$ and StateDiagramMM, where the StateDiagram metaclass extends the Behavior metaclass coming from $A_0$, thus it has also a name, a description and can be associated to all the architectural elements that have a behavioural description, e.g., SComponent and SInterface. Moreover, State and Transition extend the PropertyHolderElement metaclass of $A_0$ so that software architects are allowed to associate properties to them. In the remainder of this case study I refer to the extended version of $A_0$ when applying DUALLY for architecture models interoperability.

**DIA.SPEC**

DiaSpec [20] is a declarative language for describing applications that orchestrate entities interacting with an environment. In research, those applications are called Sense-Compute-Control (SCC) applications (they are also referred to as pervasive computing systems) [142]. As shown in Figure 5.8, an SCC application is composed of three main types of actions: sensing the environment to get the needed information, perform some computation on the data gained from the environment, and controlling the environment by performing some operation on it. Under this perspective, DiaSpec is a textual language for describing the architecture of SCC applications. The following listing shows the grammar of Diaspec.  

```plaintext
1  -- Root
2  document: ( device_def | context_def | controller_def | action_def | struct_def | enum_def )* ;
3  -- Definitions
4  device_def: "device" CID ("extends" CID)? '{' ( attribute_def | source_def | action_impl)* '}' ;
```
A DiaSpec specification is composed of two main parts: the taxonomy description and the application description.

- **Taxonomy description.** It described the types of *devices* that will be used by the application. A device consists of sensing capabilities (i.e., producing data via *sources*), and actuating capabilities (i.e., providing *actions*). Each device may have a set of *attributes* containing specific properties of the device; the type of an attribute may be either *primitive* (e.g., an integer), or *structured* (e.g., a structure, or enumeration). Device declarations are organized hierarchically, allowing device types to inherit attributes, sources and actions.

- **Application description.** An application senses the environment via devices, in-
5.2 Architecture Framework

Introduced in the taxonomy description. The resulting data are refined by context components to match the application needs. A context component has a list of context/source names as input, and produces a (primitive or structured) type as output. The data produced by a context component may be relative to some situation (e.g., the physical location of a component), in this case the data is associated to an index that helps in identifying the associated situation. Context data are then passed to controller components to make decisions by triggering device actions, declared in the taxonomy. Thus, a context component can take a list of context information as input and produces actions as output.

DiaSpec is supported by a dedicated tool called DiaSuite. Fundamentally, DiaSuite is an integrated development environment that covers the whole development cycle of the development of SCC applications. More specifically, it provides a customized support for each development phase. Indeed, the design phase is supported by the DiaSpec architecture description language; the implementation phase is supported by the Java source code automatically generated from the specification; the testing phase is supported by a 2D simulator leveraging the DiaSpec specification that allows developers to interactively simulate the model defined in DiaSpec; the deployment phase is supported by backends based on distribution platforms like SIP, RMI, etc. Diasuite allows developers to be confident about the conformance between the design and the implementation of the system by relying on generative programming techniques.

The MEGAF infrastructure operates on metamodels, thus in order to import DiaSpec into MEGAF we need its corresponding metamodel. I analysed the DiaSuite tool, and I found out that an Ecore metamodel is already available in the source code of DiaSuite. This is because the DiaSuite backend is based on EMF, and the DiaSpec textual specifications are actually serialized as standard EMF models. Once I obtained such a metamodel, it must be imported into MEGAF: so it has been tagged as ADL since it contains architectural information, then it has been semantically linked to $A_0$.

Since we are planning to being able to transform DiaSpec models towards both UML and FSP models, I linked the DiaSpec metamodel and $A_0$ by means of the Dually weaving model. By doing this, the DiaSpec-$A_0$ weaving model (whose extract is shown in Figure 5.9) will drive the generation of model transformations between DiaSpec and $A_0$. The DiaSpec-$A_0$ weaving model contains an equivalence correspondence between Document and SoftwareArchitecture since they both represent a container for architectural elements. Also, the DiaSpec-$A_0$ weaving model defines equivalence correspondences from DeviceDef, ContextDef and ControllerDef to SACOMPONENT with a set of conditions that will help to distinguish which elements an SACOMPONENT corresponds to when transforming from $A_0$ back to DiaSpec. SourceDef and ActionImpl can be considered as a kind of interface of the elements previously mapped to SACOMPONENT, so they are mapped to SINTERFACE via equivalence correspondences (with a set of conditions similar to the correspondences between DeviceDef and SACOMPONENT), whereas ActionDef is mapped to SATYPE. ContextRef, SourceRef and ActionRef are linked to SACHannel in

---

4http://diasuite.inria.fr
We are planning to use DiaSpec for modeling the structure of the BusOnAir system, with a specific focus on its Bus sub-system. Moreover, since DiaSpec is a specific language for system obeying to the Sense-Compute-Control architectural style, it nicely fits with our concerns regarding the Bus sub-system, however we faced some difficulties in using it also for modeling the Central Station and Mobile App sub-systems since they do not follow the SCC architectural style. We managed to overcome this limitation by using BYADL to extend DiaSpec for describing also generic components; this part of the case study is described in Section 5.3. The usage of the resulting extended language is described in Section 5.4.1.
UML

For what concerns UML, we reuse a dedicated UML profile for specifying component-based architectures in terms of components, connectors and their behaviour. The profile is called UMLcc and it has been introduced in [106]. UMLcc is designed having in mind the guidelines provided in [78] and extending them in order to describe also connector roles, behaviour and hierarchically-structured systems.

As shown in Figure 5.10, according to the UMLcc profile a software architecture is described by a component diagram containing the main components (represented by the CcComponent-Instance stereotype), connectors (CcConnector-Instance) and their configuration. CcComponent-Instances interact with their external environment through a set of CcPorts, while CcConnector-Instances communicate via CcRoles; they both extend the UML port metaclass. Both Components and connectors communicate through CcAttachments (i.e. stereotyped UML Dependencies) representing generic communication channels. CcDelegations are stereotyped UML delegation connectors that link hierarchically structured components. Behaviour is expressed in terms of state machines (CcBehav-vior) and can be specified for both components and connectors. An optional auxiliary component diagram specifies the type system of the architecture being modelled in terms of CcComponentTypes and CcConnectorTypes.

By using the automatic bridge between UML profiles and Ecore metamodels described in Section 4.2.1, the UMLcc profile can be automatically transformed into its corresponding UMLcc metamodel, which contains the same concepts. The resulting metamodel is then tagged as ADL since it represents architectural concepts.

At this point the UMLcc metamodel must be linked to $A_0$. Since we are planning to
being able to transform between UML\textsubscript{cc} models and FSP models, and since I already used UML\textsubscript{cc} in DUALLY for a past work (see [106]), I decided to reuse the link between UML\textsubscript{cc} and \(A_0\) already developed in DUALLY. By doing this, the \(UML_{\text{cc}} - A_0\) weaving model containing the correspondences between the UML\textsubscript{cc} profile and the \(A_0\) metamodel, will drive the generation of model transformations between UML\textsubscript{cc} and \(A_0\).

As shown in Figure 5.11, \(UML_{\text{cc}} - A_0\) defines an equivalence correspondence between CcComponentInstance and SAcomponent, whereas CcConnectorInstance corresponds to SAconnector. Both CcPort and CcRole correspond to SAinterface in \(A_0\). CcAttachment in UML\textsubscript{cc} corresponds to SAchannel in \(A_0\) and vice versa, while CcDelegation corresponds to SAbinding in \(A_0\) and vice versa. Both CcComponentType and CcConnectorType correspond to SAstructuredType in \(A_0\). CcBehavior is mapped into StateDiagram in \(A_0\) and vice versa. State from UML corresponds to State in \(A_0\) and vice versa. Transition from UML is mapped into a Transition in \(A_0\). Each UML initial PseudoState corresponds to an \(A_0\) InitialState and vice versa, whereas a UML FinalState corresponds to a FinalState and vice versa. The interested reader may refer to [106] for more details on both UML\textsubscript{cc} and its weaving model towards \(A_0\).

In this case study I use UML for two main purposes: (i) to describe the various components of the BusOnAir system with a widely known language, and (ii) to describe the behaviour of each component by means of state machines. In Sections 5.4.1 and 5.4.2 I will go into the details on this part of the case study.
FSP

FSP [98] stands for Finite State Process and it is the notation introduced by Jeff Magee and his team for specifying the behaviour of concurrent systems. In the past work presented in [106] FSP has been used in DUALLY, together with another architectural notation called Darwin [99]. FSP models behavioural aspects of a software system in terms of concurrent processes; the LTSA tool in turn generates a labelled transition system from the FSP description. Designers can animate the LTS by stepping through the sequences of actions, and model-check the LTS for various properties (e.g., deadlock freedom and safety). Two fundamental elements are present in an FSP specification: actions (which make up processes) and states. Actions are either input actions that come from the process’s environment, or output actions that originate from the process. Processes can be described recursively and may be composed in parallel.

![The FSP metamodel](image)

The FSP metamodel shown in Figure 5.12 is an extract of the DarwinFSP metamodel defined in [106]. We constructed the joint Darwin/FSP metamodel directly encoding the two official BNFs5 into their corresponding metamodel. Also, we defined a text-to-model transformation and a model-to-text transformation to transform Darwin and FSP textual specifications into a model conforming to the FSP metamodel, and vice versa. A model conforming to the FSP metamodel is composed of a set of ProcessDeclarations. Each ProcessDeclaration is made up of PrefixableElements which can in turn be: (i) simple Actions; (ii) GuardedActions which carry an activation guard; (iii) Choices which carry a choice statement and (iv) ProcessCalls which embody a call to another ProcessDeclaration. FSP semantics also imposes the existence of a relabeling mechanism represented

---

by the Relabel element, used only with relabelling operations (i.e., operations which deliberately change the label within the specification). Just like its Darwin counterpart, FSP provides the possibility of defining Constants to be valid throughout the specification. It must be noted that the presence of the three special Processes, namely STOP, ERROR, END, represent three special end-points, which must be present in the FSP specification by default.

The FSP metamodel is then tagged as ADL since it represents architectural concepts.

![Diagram](image)

**Figure 5.13:** DULLY weaving model between DarwinFSP and A₀

Figure 5.13 shows the original DarwinFSP_A₀ weaving model defined in [106]. According to this weaving model, a Darwin ComponentInstance corresponds to an SAcomponent of A₀ and vice versa. Also SAconnector in A₀ is mapped into a Darwin ComponentInstance. This is due to the fact that Darwin does not have the concept of software connector and the semantically closest concept to it in Darwin is ComponentInstance. A Portal corresponds to an A₀ SAinterface and vice versa. The kind of SAinterface is set according to the type of the Darwin/FSP Portal (e.g. if the portal is “provide” or “require” then the SAinterface direction is “Output” or “Input”, respectively). ComponentDeclaration and Binding in Darwin/FSP correspond to SAchannel and SASTructuredtype in A₀, respectively. Each DarwinFSP ProcessDeclaration is mapped to a StateDiagram and vice versa. ActionPrefix and Choice are both mapped to an A₀ State and vice versa. Darwin/FSP Action is mapped to Transition and vice versa. ERROR, END and STOP can be considered as special kinds of final states, so we mapped them to A₀ FinalState. Since we will not use the Darwin notation, in the remainder of this case study we will not consider the portion of weaving model related to Darwin.
In this case study I use FSP for formally representing the behaviour of the components defined in UML, and then to perform dependability and deadlock freedom analysis on it. In Section 5.4.2 I will go into the details on this part of the case study.

**RESTLANG**

The term REST is the acronym of Representational State Transfer [51] and it is a style of software architecture for distributed hypermedia systems, such as the World Wide Web. The conceptual foundation of REST has been defined by Roy Thomas Fielding in 2000 and since then it received a lot of attention both from academia and research. Indeed, most of the current web applications (e.g., Youtube, Google Maps, Twitter, Digg, etc.) provide their services by means of a REST interface. A RESTful web service is also called a RESTful web API and it realizes a simple web service implemented using HTTP and the principles of REST.

Fundamentally, REST allows developers to interact with a resource on the web by means of a simple, plain URL. The set of operations supported by REST are the classical HTTP methods: GET, PUT, POST, and DELETE. Resources can be accessed by means of GET request, which is the most common action used in REST. In the current state of the practice a lot of APIs only allow to read data from their services, these APIs just need to implement the GET operation only. A POST operation is used to add data to the service, and possibly then to their database. The other two operations associated with REST are PUT and DELETE. The first one allows a user to replace some data with newly available data in the service, whereas the second one allows a user to delete some data from the service. A REST call can also have some output values; the most used ones are: XML, RSS, JSON, or CSV. For a further discussion on REST, the interested reader may refer to [51].

Using a REST interface provides a wide series of benefits both to the interface developers and to the interface users, among the others:

- REST interface are simple to maintain and to access;
- users (i.e., developers mainly) are very familiar with the format of REST requests since they are standard HTTP methods;
- HTTP cache and proxy servers can be used to help designers handle high workload since the REST interface is based on standard HTTP methods;
- it helps designers to manage even complex applications since they can be organizes into simple (REST) resources;
- the REST API provides a uniform interface for heterogeneous applications accessing them; i.e., applications obeying to different paradigms, written in different programming languages and paradigms, they all access the REST interface in the same way;
results of REST calls are human readable.

Since in BusOnAir we want to take advantages of the benefits of REST, we decided to realize the communication between the Central Station system and both the Bus and Mobile App systems as a RESTful web service. We decided to model our REST web services by means of a dedicated modeling language. Unfortunately, to the best of our knowledge there is not a dedicated language for describing REST services in a simple and effective manner. So, we decided to reuse an already defined metamodel for REST services [145], and we reuse the standard EMF modeling editor for creating models conforming to such a REST metamodel. I call the resulting modeling language RESTLANG. Figure 5.14 shows an overview of the REST metamodel defined in [145].

Figure 5.14: The REST metamodel

**RESTService** is the root element of each model conforming to the REST metamodel described above. Each REST service is an aggregation of different **Resources** that can be reached from a base URI (Universal Resource Identifier). Each **Resource** represents an entity coming from the domain of interest (e.g., a photo, a song, a book, etc.) and its state is exposed and can be changed accessing to a specific URI. A REST Service may include global arguments that must be always specified in the requests (e.g., an authentication key for identifying the client that is performing the request). Global arguments are represented by a set of **Parameters**; they can have a **DataType** (int, string, float, etc.), a boolean attribute to specify if a parameter is **required** or not, a **default** value to be sent, and the **inQueryString** attribute to set if the parameter value is sent to the resource encoded in the URI query string. A **Representation** is the description of the messages sent or received from a Resource in terms of a technological language. The most used languages
for describing representations are XML and JSON; an XMLRepresentation specifies an XML Schema for defining its syntax, and XSDElement from this schema to describe a specific Resource representation; a JSONRepresentation contains a string encoding the Javascript object that it is representing. A Method is the means by which a specific resource can be accessed. Fundamentally, it provides an abstraction layer upon the sending of a request-response message using a HTTPMethod (GET, POST, PUT or DELETE). A method can have also other attributes. The isDefaultInvoke specifies whether the method will be performed by default when the resource is accessed or not. The path attribute specifies the path that must be appended to the URI of the resource in order to access that specific method. A Method is made up of a least one Request and a set of Response messages. The Request can include a set of additional Parameters to be sent to the resource. If more complex data structures must be sent, the Request parameters can be described using a Representation. A Response message returns the state of the resource after a method invocation and it references at least a representation. A special kind of response is the ErrorResponse, it describes faulty, incorrect responses and can provide a standard HTTP error code (HTTPCode in the metamodel). The interested reader may refer to [145] to have a complete description of the REST metamodel, and an example of a model conforming to it can be seen in Section 5.4.3.

For what concerns the semantic links to $A_0$, since the REST metamodel fundamentally does not contain any relevant architectural information, we simply linked every root metaclass in the RESTLANG metamodel to the root element of the $A_0$ metamodel, that is Element. In the same line, we did not assign any tag to the REST metamodel when importing it into MEGAF.

### 5.2.4 Correspondence Rules

Before going into the description of the correspondence rules that may be part of the architecture framework under study, it should be noted that MEGAF already provides a built-in mechanism for performing some checks. Indeed, MEGAF contains a set of rules that help architects in establishing if the current architecture framework conforms to the ISO/IEC/IEEE 42010 standard. For example, those rule can check if each system stakeholder concern is framed by at least a viewpoint, or if by mistake there is some stakeholder who does not have any concern associated to it, etc. In MEGAF those kind of checks are already defined for all architecture viewpoints and, since they have been formalized as OCL constraints running on the MEGAF megamodel, their execution is a one-click action for the architect.

Next step is the definition of a set of correspondence rules that help architects in enforcing project- or system- specific relations that must hold when creating the architecture description. In MEGAF correspondence rules can be formalized by using the OCL constraint language and can be added to the MEGAF megamodel with a dedicated wizard; it is up to the MEGAF engine to check which correspondence rules have been added to the MEGAF megamodel and to execute them when needed. As a representative sample, below
I provide some of the correspondence rules that have been defined for the BusOnAir case study:

- **DIASUITECOMP_REST**: each resource defined in a REST model in the WebServices view must be associated to at least a DiaSpecComp component in the System view. This correspondence rule governs correspondences that must be defined in the architecture description conforming to this architecture framework. These correspondences must be manually defined by the software architect as weaving models, and can be identified by MEGAF because they have the same name of the correspondence rule. Listing 5.2 shows how this correspondence rule has been defined as an OCL constraint.

Listing 5.2: DIASUITECOMP_REST correspondence rule as an ATL query.

```java
query DIASUITECOMP_REST = 

if RESTLANG! Resource . allInstancesFrom( 'IN' ) - > exists ( e | not e .
  isLinkedToComponent( ) )
then
  'DIASUITECOMP_REST is violated, violating resources: ' +
  RESTLANG! Resource . allInstancesFrom( 'IN' ) - > select ( e | not e .
    isLinkedToComponent( ) ) . toString()
else
  'DIASUITECOMP_REST is not violated.'
endif . println();

helper context RESTLANG! Resource def isLinkedToComponent( ) : Boolean =
  thisModule . getCorrespondenceByName( 'DIASUITECOMP_REST' ) . links - > exists ( e | e .
    left . getReferredElement( ) = self
        and e . right . getReferredElement( ) . isKindOf( DIASUITECOMP! Component ) ) ;

helper def getCorrespondenceByName( name : String ) : AM3! WModel =
  AM3! WeavingModel . allInstancesFrom( 'MEGAMODEL' ) - > select ( e | e . name = name );
```

Listing 5.2 is composed of an ATL query that checks in line 2 if each REST resource in the RESTLANG model (referred by the 'IN' identifier), is actually linked by means of a weaving link to a UML component in the UML model. If this check does not evaluate to true, then the correspondence rule is violated (line 3 in the listing), and all the REST resources violating it are printed to the console (lines 4 and 7 in the listing). The query makes use of two auxiliary helpers. The first helper (lines 9-11 in the listing) is called `isLinkedToComponent`; it navigates the weaving model called with the same name of the rule (i.e., DIASUITECOMP_REST) and checks if the given REST resource is linked to a UML component. The second helper (line 13-14 in the listing) is called `getCorrespondenceByName` and it checks if in the MEGAF megamodel there is a correspondence (that is, a weaving model) with `name` as name.

- **UML.UML**: in the System view, each UML component must contain a UML state machine describing its behaviour. This is an example of intra-model correspondence rule, and it states in a reusable way a constraint that must hold within a single architecture model. This case is interesting because this kind of correspondence rules do not need to govern a correspondence defined in the AD, they simply predicates on a single architecture model (a UML model in this case). Listing 5.3 shows how this correspondence rule has been defined as an OCL constraint.
Listing 5.3: UML_UML correspondence rule as an ATL query.

```java
query UML_UML =
if (UML! Component . allInstancesFrom ('IN') -> exists (e | e.ownedBehavior -> select (e2 | e2 . isTypeOf (UML!StateMachine ). isEmpty ()))) then
  'UML_UML is violated , violating resources: ' +
UML! Component . allInstancesFrom ('IN') -> select (e | e.ownedBehavior -> select (e2 | e2. isTypeOf (UML!StateMachine). isEmpty ()) ). toString ()
else
  'UML_UML is not violated .' endif . println ();
```

Listing 5.3 is composed of an ATL query that checks in line 2 if each UML component in the UML model (referred by the 'IN' identifier) contains a state machine as ownedBehavior. If this check does not evaluate to true, then the correspondence rule is violated (line 3 in the listing), and the UML components violating it are printed to the console (lines 4 and 7 in the listing).

• UML_FSP: in the Behaviour view, for each UML state machine in the System view there must be a process definition with the same name in the FSP model. This is an example of correspondence rule that relates two different architectural models in the same view. Listing 5.4 shows how this correspondence rule has been defined as an OCL constraint.

Listing 5.4: UML_FSP correspondence rule as an ATL query.

```java
query UML_FSP =
if (not UML! Component . allInstancesFrom ('IN') -> exists (e | thisModule . getFSPProcessByName (e . name))) then
  'UML_FSP is violated , violating resources: ' +
UML! Component . allInstancesFrom ('IN') -> select (e — thisModule . getFSPProcessByName (e . name)). toString ()
else
  'UML_FSP is not violated .' endif . println ();
```

Listing 5.4 is composed of an ATL query that checks in line 2 if for each UML component in the UML model (referred by the 'IN' identifier), exists an FSP process definition with the same name. If this check does not evaluate to true, then the correspondence rule is violated (line 3 in the listing), and the UML components violating it are printed to the console (lines 4 and 7 in the listing). The query makes use of an auxiliary helper that returns a process definition with name name from a given FSP model referred by the 'IN2' identifier.

• DIASUITECOMP_UML: in the System view, for each generic component in the DiaSpecComp model there must be a UML component with the same name. This is another example of correspondence rule that relates two different architectural models in the same view. Listing 5.5 shows how this correspondence rule has been defined as an OCL constraint.

Listing 5.5: DIASUITECOMP_UML correspondence rule as an ATL query.

```java
query DIASUITECOMP_UML =
if (not DIASPECCOMP! Component . allInstancesFrom ('IN') -> exists (e | thisModule . getUMLComponentByName (e . name))) then
```

Listing 5.5 is composed of an ATL query that checks in line 2 if for each UML component in the UML model (referred by the 'IN' identifier), exists a UML component with the same name. If this check does not evaluate to true, then the correspondence rule is violated (line 3 in the listing), and the UML components violating it are printed to the console (lines 4 and 7 in the listing).
Chapter 5. The BusOnAir Case Study

```java
3 'DIASUITECOMP UML is violated, violating resources: ' +
4 DIASPECCOMP!Component . allInstancesFrom ( 'IN' ) -> select ( e | thisModule.
getUMLComponentByName(e . name) ). toString ()
5 else
6 'DIASUITECOMP UML is not violated. '
7 endif . println ();
8
9 helper def : getUMLComponentByName(name : String ) : UML! Component =
10 UML! Component . allInstancesFrom ( 'IN2' ) -> any(e | e . name = name) ;
```

The behaviour of the listing above follows the same logic of Listing 5.4.

5.3 EXTENSION OF DiaSpec

As anticipated in Section 5.2.3, DiaSpec nicely fits with our concerns regarding the Bus sub-system, however we faced some difficulties in using it also for modeling the Central Station and Mobile App sub-systems since they do not follow the Sense-Compute-Control architectural style. In this section I will describe how BYADL has been used to extend DiaSpec for describing also generic components.

First of all, a suitable metamodel for representing generic components must be defined, and then imported into the MEGAF megamodel: we call such a metamodel ComponentMM. It is imported into the MEGAF megamodel by means of the standard importing procedure and it is tagged as Extension since it will be used for extending an already existing language (i.e., DiaSpec). It is important to note that the ComponentMM metamodel is now part of the MEGAF megamodel, and thus it can be freely reused by architects for future extensions or customizations of existing ADLs. The semantic links to A0 are: Component is linked to Component in A0, Port is linked to InteractionPoint in A0, and Connection is linked to Link in A0. Figure 5.15 shows the ComponentMM metamodel.

![Figure 5.15: The componentMM metamodel](image-url)
As can be notice in the figure, the ComponentMM metamodel is very simple and generic. I did it by purpose for two main reasons. Firstly, since in BYADL extension metamodels can be reused for extending different ADLs and in the context of different architecture frameworks, I tried to be as generic as possible; secondly, having a simple extension metamodel helps the reader in better understanding the composition mechanism of BYADL. The ComponentMM metamodel is composed by three metaclasses: Component, Port and Connection. The Component metaclass represents computational entities that encapsulate a set of related functions. Those functions are represented by Ports that can be connected to other ports by means of a Connection. All metaclasses within the ComponentMM metamodel can have an associated name to identify their instances. Furthermore, each component can be organized hierarchically, that is, it can contain other components and connections. Finally, the ComponentMM metamodel comprises also an OCL constraint stating that each instance of component can contain only instances of the Component metaclass, and not one of its (future) specializations. This OCL rule does not pose any relevant constraints on instances of the current ComponentMM metamodel, but it will turn out to be useful when combining the DiaSpec metamodel and ComponentMM.

Figure 5.16 shows how the DiaSpec metamodel has been composed with the ComponentMM metamodel in order to extend the DiaSpec language for describing also generic components. Such a composition is formalized as a weaving model called DiaSpecExtension; it contains the applications of the composition operators as described in Section 3.3.1.

![Figure 5.16: Composition of DiaSpec and ComponentMM](image)

According to the DiaSpecExtension weaving model, the Inherit operator has been applied three times. Those applications of the Inherit operator specify that the DiaSpec concepts of ControllerDef, ContextDef and DeviceDef will be subtypes of the Component metaclass from ComponentMM in the resulting extended metamodel (which I hereby call DiaSpecComp). For the sake of simplicity, I do not show here the resulting extended metamodel DiaSpecComp. Basically, it contains all the metaclasses from the DiaSpec and ComponentMM metamodels, where ControllerDef, ContextDef and DeviceDef are subtypes of Component; it contains also the OCL constraint defined in ComponentMM. It is interesting to note that now the extended DiaSpecComp metamodel provides the
Chapter 5. The BusOnAir Case Study

following capabilities:

1. it allows architects to define basic DiaSpec specifications in the same way as the original DiaSpec language does;
2. it allows architects to model also generic components within a DiaSpec specification, and to link them by means of generic connections among generic ports;
3. it allows architects to consider the various devices, controllers and contexts of a basic DiaSpec specification, and to detail their internal structure by means of generic components, ports, and connections.

For each application of the Inherit operator, the abstractTarget attribute is set to false, so that the Component metaclass in the resulting extended metamodel will not be abstract; this gives a flavour of the flexibility of the operators proposed in BYADL, indeed if the abstractTarget attribute was set to true, the concept of generic component could not be used within models conforming to the resulting extended metamodel.

At this point, the objective of the OCL constraint in the original ComponentMM metamodel should be clear: it forces the architect to define the inner elements of either a device, context or controller as generic components (and connections) only. Putting it in another way, the OCL constraint precludes the definition of either devices, controllers or contexts inside other devices, controllers, or contexts.

Once the extended DiaSpecComp metamodel has been generated, we further use BYADL to perform the following tasks:

- the DiaSpecComp metamodel is automatically tagged as ADL because it contains architectural concepts;
- the DiaSpecComp metamodel is automatically registered into the MEGAF megamodel;
- the semantic links between the DiaSpecComp metamodel and $A_0$ have been automatically generated;
- tree-based, textual and graphical syntaxes have been automatically generated;
- the DiaSpecComp2DiaSpec model migrator is automatically generated (it allows to transform models conforming to DiaSpecComp to models conforming to DiaSpec);
- the DiaSpecComp2DiaSpec model migrator is automatically generated (it allows to transform specifications conforming to DiaSpec to models conforming to DiaSpec-Comp);

Both the graphical notation and the model migrators of the last two points will be described and used in Section 5.4.1 in the context of a concrete architecture model of BusOnAir.
5.4 Architecture Description

In this section I will describe how the just created architecture framework has been used to define an architecture description of the BusOnAir system. Fundamentally, each viewpoint defined in the framework will be instantiated into an architecture view, and the architecture models pertaining to each view will be defined. Figure 5.17 shows the used architecture views and their architecture models.

Figure 5.17: Architecture views for the BusOnAir architecture description

It is interesting to note that, in accordance with the ISO/IEC/IEEE 42010 standard, an architecture view can contain more than one architecture models (e.g., the System view contains three models conforming to DiaSpec, DiaSpecComp and UML). Also, a single architecture model can be shared between different views; this is the case of the UML in BusOnAir that is shared between the System and Behaviour views (indeed it is well-known that a UML model can contain both structural and behavioural concepts). Each view of the BusOnAir architecture description will be described in the next sections.

Furthermore, MEGAF provides a built-in mechanism for performing some checks on the defined architecture views. Indeed, MEGAF contains a set of rules that help architects in establishing if the current architecture description conforms to the architecture framework defined for BusOnAir. For example, those rule can check if each stakeholder concern is addressed by at least an architecture view, if an architecture view does not contain any architecture model, etc. These checks are possible thanks to the megamodel underlying MEGAF since it allows to keep track of all the various relationships between the architectural elements within the megamodel and to predicate on them by means of generic conformance rules. Figure 5.18 graphically renders the workflow we followed for creating the architecture description of BusOnAir. In the following I will describe each step of the workflow.
Figure 5.18: Workflow for architecting the BusOnAir system

1. **Creation of the DiaSpec model**: a model of the Bus System of BusOnAir is created using DiaSpec.

2. **Generation of the DiaSpecComp model**: the DiaSpecComp2DiaSpec migrator generated by BYADL is now used to automatically obtain a DiaSpecComp model from the previously defined DiaSpec model.

3. **Enhancement of the DiaSpecComp model**: the DiaSpecComp model is now used as a starting point for creating a complete model of the whole BusOnAir system. So in this step the DiaSpecComp model is completed with a description of also the Central Station and Mobile App subsystem.

4. **Generation of the UMLcc model**: two model transformations previously generated by DUALLY (see Section 5.2.3) are executed in order to automatically generate a UMLcc model from the current DiaSpecComp model.

5. **Behaviour definition in UMLcc**: the current UMLcc model is manually augmented by specifying a UML state machine for each component of the BusOnAir system.

6. **Generation of the FSP specification**: the model transformations previously generated by DUALLY (see Section 5.2.3) are executed in order to automatically gen-
erate an FSP model containing a formal specification of the behaviour of the BusOnAir system.

7. **FSP model analysis**: the FSP specification previously obtained is used to perform liveness and deadlock freedom analysis on the behaviour of the BusOnAir system.

8. **Definition of the RESTLANG model**: a model representing all the REST services exposed by the Central Station subsystem is defined. It will be also linked to the DiaSpecComp model by means of an architectural correspondence that will be governed by the DIASPECCOMP_REST correspondence rule described in Section 5.2.4.

It is clear in Figure 5.18 that the combination of **MEGAF**, **DUALLY** and **BYADL** provides a reasonable level of automation and helps designers in keeping a consistent network of architecture models with low additional effort. Figure 5.19 shows a refined view of the workflow we followed for creating the BusOnAir system, in which I highlight the various model transformations generated by either **DUALLY** and **BYADL**, they are:

- **DiaSpec2DiaSpecComp**: it is a model migrator automatically generated by **BYADL** from the weaving model that has been used to extend the DiaSpec metamodel with the ComponentMM metamodel. It takes as input a DiaSpec model and produces as output an instance of the DiaSpecComp metamodel accordingly to the operators that have been applied during the metamodels composition. Actually, **BYADL** can generate also the inverse transformation (i.e., DiaSpecComp2DiaSpec), but I did not show it here because we never used it in this case study.

- **DiaSpecComp2A0**: it is automatically generated by **DUALLY** from the weaving model that semantically links the DiaSpec metamodel to **A0** (see Section 5.2.3. It takes as input a DiaSpec model and produces as output a model conforming to **A0**.

- **A02UML**: it is automatically generated by **DUALLY** from the weaving model that semantically links the UMLcc metamodel to **A0** (see Section 5.2.3. It takes as input a model conforming to **A0** and generates a UML model on which the UMLcc profile has been applied.

- **UML2A0**: it is automatically generated by **DUALLY** and performs the inverse task of the **A02UML** transformation.

- **A02FSP**: it is automatically generated by **DUALLY** from the weaving model that semantically links the FSP metamodel to **A0** (see Section 5.2.3. It takes as input a model conforming to **A0** and generates an FSP specification.

- **FSP2A0**: it is automatically generated by **DUALLY** and performs the inverse task of the **A02FSP** transformation.
It is important to note that the execution of those transformations is totally transparent to the architect. Indeed, when executing the \textit{DiaSpecComp2DiaSpec} transformation generated by BYADL, the architect simply tells that (s)he wants to pass from DiaSpec to DiaSpecComp, then it is up to the BYADL engine to select and execute the needed model migrators to accomplish this task. Moreover, when an architect wants to transform an architecture model from a language to another, the \texttt{DUALLY} model transformations are automatically selected and executed by the \texttt{DUALLY} engine; also, the architect does not even need to know that behind the lines \texttt{DUALLY} is passing through the intermediate $A_{0}$ language.

In the following sections I will give a high-level description of each architecture view defined and the various architecture models involved in this case study. Under this perspective, the description of the involved models is very abstract since the focus here is more on the mechanisms for creating the architecture framework and for relating the architecture description languages, rather than on the semantic meaning of the models themselves.
5.4 Architecture Description

5.4.1 **System View**

**STEP 1.** As anticipated in Section 5.2.2, the *System viewpoint* is used to break down the BusOnAir system into a set of components, and to show how they relate to each other. Under this perspective we decided to firstly describe the Bus System with a dedicated language called DiaSpec, and then to use it as starting point for creating a complete model of the whole BusOnAir system.

In this initial phase, we use the DiaSpec language because it fits with the sense-compute-control nature of the Bus system, and because it allows (i) to generate skeleton Java code for each part of the model and (ii) to interactively simulate the architecture model in order to quickly check if behaves like expected. Listing 5.6 shows the DiaSpec specification as it is defined in its corresponding tool.

```plaintext
// TAXONOMY
// DEVICES
device RFIDReader {
    source idTraveller as String;
}
device Network {
    source paymentData as Integer;
    action StartTrip;
    action Pay;
    action SendBusLocation;
}
device GPSReceiver {
    source position as String;
}
device Alarm {
    action Activation;
}
device Screen {
    action Display;
}

// ACTIONS
action Activation {
    activate();
    deactivate();
}
action Display {
    display(message as String);
}
action Pay {
    sendPaymentRequest(amount as Integer);
}
action StartTrip {
    location(location as String);
}
action SendBusLocation {
    location(location as String);
}

// APPLICATION ARCHITECTURE
// CONTEXTS
context ReadIDIn as String {
    source idTraveller from RFIDReader;
}
context ReadIDOut as String {
    source idTraveller from RFIDReader;
}
context MoneyChecker as Integer {
    source paymentData from Network;
}
```
Fundamentally, the architecture model can be broken down to two main parts: a taxonomy of the used devices, and the actual architecture of the application realizing the Bus System. The taxonomy defines five devices that will be deployed on the Bus, they are:

- **RFIDReader**: the RFID reader that allows users to check in and check out when either entering and leaving the bus. The reader reads the card and makes the id of the traveller available to the other software components within the model.

- **Network**: the network device used by the Bus system to communicate with other subsystems within the BusOnAir project.

- **GPSReceiver**: it receives GPS data from a satellite and communicates the actual location of the bus to the other components of the system.

- **Screen**: it serves to show messages to an LCD display.

- **Alarm**: when activated, it emits a sound and a light in order to catch the attention of the driver.

Some of the above mentioned devices provide raw data to other components called contexts, they are:

- **ReadIDIn**: it gets the id of the traveller who entered the bus, and refines it into a set of additional information that will be used by the *TripManager* controller.

- **ReadIDOOut**: it gets the id of the traveller who is leaving the bus, and refines it into a set of additional information that will be used by the *TripManager* controller.

- **MoneyChecker**: it manages the result of the payment of a trip, and tells to the *TripManager* controller if the trip has been regularly paid or not.
• CurrentLocation: it gets raw GPS data from the \textit{GPSReceiver} device and refines it into a couple of geographic coordinates; then it sends those coordinates to the \textit{BusLocationManager} controller.

Most of the logic of the Bus sub-system resides in the two controllers of this model: \textit{TripManager} and \textit{BusLocationManager}. \textit{TripManager} mainly performs three main tasks: (i) each time a traveller starts a trip, it records a set of information about the starting trip and send it to the central station by means of the Network device, it also show a "welcome message" via the \textit{Screen} device, (ii) when the traveller is leaving the bus it sends a payment request to the central station via the Network device, and (iii) according to the result of a previously sent payment request, it either shows a "goodbye message" to the traveller or activates the \textit{Alarm} device. \textit{BusLocationManager} gets the coordinates of the current location of the bus and sends them to the Central Station system by means of the \textit{Network} device. Those coordinates then will be used to update the routes database in the central station.

Figure 5.20 shows a screenshot of the simulation tool for DiaSpec in which the system has just discovered a "bad traveller", i.e., a traveller whose payment request did not finish regularly.

![Figure 5.20: Simulation of the Bus system in Diasuite](image)

STEP 2. At this point, we will model the whole BusOnAir system by using the above described model as starting point. In this context, we use the \textit{DiaSpec2DiaSpecComp} model migrator generated by BYADL in order to have an initial DiaSpecComp model to work on.

Figure 5.21 shows the initial DiaSpecComp model. Looking at the figure we can notice some interesting aspects. Firstly, we automatically obtained a graphical editor for DiaSpec; this is thanks to the BYADL mechanism for managing the concrete syntaxes of languages described in Section 3.3.2. The graphical conventions of this model are those
Fundamentally, all devices, contexts and controllers have been graphically rendered as plain components; actions and sources have been rendered as component ports (white ports represent input ports, black ports represent output ports, and grey ports represent input/output ports), their interconnections have been rendered as channels between ports. Secondly, we are not making the DiaSpecComp model from scratch, but we are starting from all the components that were already defined in the initial DiaSpec model. This is thanks to the model migrators semantics which always try to preserve as much information as possible when bridging between a base model and a composed one.

**STEP 3.** In this step we work on the just obtained DiaSpecComp model and we add the parts that are missing from a complete description of the whole BusOnAir system: the ones related to the Central Station and Mobile App systems. Figure 5.22 shows the revised model. The coloured components in the figure are the ones that have been added in this specific step. More specifically, the darker components in the lower part of the figure represent the Central Station system, whereas the lighter coloured components show the Mobile App system. For the sake of brevity, in this step I focus on the description of the Central Station system, since a description of the Mobile App system does not add any relevant detail to this part of the dissertation.

The core of the Central Station system is composed of two databases. The first one is called *RoutesDB* and keeps track of all the information about the bus routes in the city, about their real-time position within the streets network, and about the number of users that are currently travelling on the buses. The second database is called *UserDataDB* and contains all the data related to the travellers, such as traveller id, their payment data, statistical data, etc. *BusOnAirService* is another fundamental component of the Central
Figure 5.22: BusOnAir model in DiaSpecComp
Station system: it exposes the information stored in RoutesDB and UserDataDB as a set of web services. Those web services can be accessed by external entities (such as our MobileApp component, or other applications), and can be considered as the data provider of our web application (WebApp in the figure).

The VehiclesPositionManager component receives the current geographical location of each bus, it computes on-the-fly an estimate of the arrival time of each bus, and stores this information in the RoutesDB; this computation is refined by interacting with the TrafficInfoManager component which in turn uses an external service for gaining traffic information in real-time. The PaymentManager manages all the payment transactions and activities performed by the travellers and accesses/stores all payment-related data into the UserDataDB.

**STEP 4.** Once we have a complete DiaSpecComp model of the BusOnAir system, we can consider its corresponding UMLcc model. This model is automatically created by DUALLY by executing the previously generated DiaSpecComp2A0 and A02UML model transformations. Figure 5.23 shows the resulting UML model.

![Figure 5.23: BusOnAir model in UMLcc](image)
5.4 Architecture Description

The obtained UML\textsubscript{cc} model, even though we passed through $A_0$, is similar to the initial DiaSpecComp model, e.g. the topography of the two models remains the same. We mapped each DiaSpecComp component to a UML component stereotyped with \textit{CCComponentInstance} since they are semantically close. Each DiaSpecComp interaction point (e.g., sources, actions, etc.) is mapped to a UML port stereotyped with \textit{CCPort}, and each information flow between interaction points is mapped to a UML dependency stereotyped with \textit{CCAttachment}.

With this step we conclude the description of the BusOnAir application from the System point of view (i.e., viewpoint), in the following steps we will firstly focus on the behavioural description and then on the definition of the web services composing the \textit{BusOnAirService} component.

5.4.2 Behaviour View

The \textit{Behaviour view} is defined to reason on how the various components of BusOnAir can be executed concurrently and to check their liveness and deadlock freedom properties.

STEP 5. In this step we consider the UML\textsubscript{cc} model and we add a UML state machine for each component in the model. By doing this, we will obtain a complete description of the BusOnAir system from a behavioural point of view. It is important to note that in this case the \textit{UML-UML} correspondence rule comes handy since it allows us to check if a state machine is missing from some component, and lets us identify which components still need to be defined with their corresponding state machine. As an example, Figure 5.24 shows (synchronous version of) the state machine associated to the \textit{TripManager} component of the Bus system.

According to the state machine, after the \textit{TripManager} is \textit{Started}, it can read (\textit{readCard} event in the state machine) the data of a card swiped on the RFID reader. At this point, \textit{TripManager} can distinguish between the situation in which the traveller is entering the bus (state \textit{RFIDReadIn}), and the one in which the traveller is leaving the bus (state \textit{RFIDReadOut}).

If the traveller is entering the bus, then \textit{TripManager} sends a \textit{showWelcomeMsg} message to the \textit{Screen} component, it records (and sends) to the Central Station a set of initial data about the starting trip (\textit{recordStartTrip} in the figure), and waits for an ”ack” from the Central Station; finally, it can come back to the \textit{Started} state and wait for another card reading event.

If the traveller is leaving the bus, then \textit{TripManager} records the current bus geographical location and sends a \textit{requestPayment} message to the Central Station. If the payment ends regularly (\textit{PaymentOK} guard in the state machine), then \textit{TripManager} clears some temporary data collected during the trip, and sends a \textit{showGoodbyeMsg} message to the \textit{Screen} component. Otherwise, if the payment does not end regularly (\textit{PaymentKO} guard in the state machine), then \textit{TripManager} fires an alarm to alert the driver about the situation,
Figure 5.24: Behaviour of the TripManager component in UML_{cc}

sends a $storeBillingData$ to the Central Station (that in turn will prepare a fine to be sent to the traveller’s postal address), and then waits that the driver stops the alarm ($stopAlarm$ event in the state machine). Finally, independently from the outcome of the payment, the TripManager component goes back to the $Started$ state and waits for another card reading event.

Since the scope of this case study is more on architecture models transformation and management, rather than on their actual semantics, I will not describe all the other state machines in our UML_{cc} model; in this context the reader can assume that a UML state machine has been defined for each component within the model.

**STEP 6.** The next step is to consider the UML_{cc} model containing both the components and the state machines describing the BusOnAir system, and to transform that model into an FSP specification. By doing this, we are able to analyse the obtained FSP model by means of its associated LTSA tool (the tool has been already discussed in Section 5.2.3). The resulting FSP specification is automatically created by DUALLY by executing the previously generated UML2A0 and A02FSP model transformations. Fundamentally, a process definition has been generated by the DUALLY transformations for each initial UML_{cc} state machine. Listing 5.7 shows the definition of the finite state process corresponding to the state machine associated to the TripManager component (see Figure 5.24).

```plaintext
1/**  BusOnAir - TripManager Behaviour  
2*   Ivano Malavolta  
3*  
4*/
5TRIPMANAGER = (start ->STARTED),  
6STARTED = (readCard ->CARDREAD | stop ->END),  
7CARDREAD = (userEntering ->RFIDREADIN | userLeaving ->RFIDREADOUT),  
8```
Listing 5.7: Behaviour of the TripManager component in FSP

As can be noted in the listing, each state in the state machine has been transformed into a process definition with the same name (the DULLY transformation has been adapted manually so that it puts in upper case the name of each state machine before transforming it into a process definition); then each transition of the state machine has been mapped into an FSP action, and if two transitions have the same source state, then they are transformed into a choice FSP element represented by the "|" symbol in the listing. More specifically, in FSP a process defined as $(a - > P | b - > Q)$ describes a process which can engage in either of the actions $a$ or $b$, and so the execution of the process may proceed in either of the two branches.

STEP 7. Once a complete FSP specification of the BusOnAir system is available, we can use the LTS toolset to (i) visualize it graphically by means of the LTS tool (Figure 5.25 for example shows the graphical representation of the TRIPMANAGER process definition described in Listing 5.7), and (ii) perform liveness and deadlock freedom analysis.

In the context of this case study the performed analyses did not discover any problem within the obtained FSP specification. More specifically, we verified that there is no risk of having deadlock between the FSP processes composing the whole BusOnAir system, and also we discovered that there are no liveness property violations. It should be noted that if some problem was discovered by the LTS tool (e.g., a deadlock between two processes in our FSP specification), then we could have to solve the problem by correcting (and thus changing) the FSP specification, and then we could have to re-apply the DULLY transformations backwards firstly to $A_0$, and then to both UMLcc and DiaSpecComp in order to propagate the changes that have been made in the FSP specification back to the other modeling languages.

In this step we also simulate interactively the behaviour of the BusOnAir system thanks to the Animator component of the LTS tool. Figure 5.26 shows a screenshot of the LTS tool animating the process definition of the TripManager component. More specifically, the screenshot shows that a traveller has already entered the bus, swiped the card to the RFID reader and the “ack” message has already been received from the Central Station; the next available actions are two: readCard that represents the fact that a traveller just swiped his card on the RFID reader, and stop that represents the turning off of the TripManager component. In general, the Animator component of LTS performs a user-controlled animation of a target FSP process. This allows architects to animate (and interactively check) large systems even if they cannot be exhaustively checked by the tool.
Figure 5.25: Behaviour of the TripManager component in FSP (graphical representation)
5.4 Architecture Description

5.4.3 Web Services View

The WebServices view is used to describe the RESTful web API exposed by the current prototype of the BusOnAirService component of the Central Station sub-system.

STEP 8. According to the workflow, the last step is the definition of a model representing the REST services exposed by the Central Station system. To this purpose, we use the REStLang modeling language described in Section 5.2.3. Figure 5.27 shows our REStLang model by means of a screenshot of the Eclipse standard EMF editor.

The figure shows that our REStLang model contains two kinds of resources: Route and Direction. Clearly, a Route represents the sequence of bus stops that compose each bus line. The Route resource exposes to methods. The first method is called getRoute and, given a certain route identifier, it returns a JSON representation of a route (i.e., the sequence of all its bus stops). In order to give a more concrete idea of the getRoute method, in the following I provide the signature of its Java equivalent method.

\[
public Route getRoute(String routeId)
\]

The second method defined for the Route resource is called allRoutes and it returns all the available bus routes stored in the RoutesDB in the Central Station system. Its Java equivalent has the following signature:
A Direction represents a portion of a traveller’s trip when (s)he is moving either by walk or by bus within the transportation network. More specifically, a direction may represent a single trip that a traveller is taking by bus, but it can also represent a short trip that the traveller is doing by walk in order to reach a specific bus stop. Direction exposes a single method called getDirections that, given two locations A and B, returns a JSON object containing the sequence of operations that a potential traveller must follow to get from a starting point A to a final destination B. The A and B points must be provided by specifying their geographical coordinates in terms of latitude and longitude; an additional parameter can be specified if the directions must be considered starting from a given initial hour of the day. The Java equivalent of the getDirections method has the following signature:

\[
\text{public List<Direction> getDirections(float fromLat, float fromLon, float toLat, float toLon, Date time)}
\]

Figure 5.27: BusOnAir REST services model in RESTLANG

In order to associate the REST services described in our RESTLANG model to the components implementing them in the System view, we need to create a correspondence between the RESTLANG model and one of the architecture models of the System view. To
this purpose, we decided to create a correspondence between the RESTLANG model to the DiaSpecComp model. In line with the MEGAF infrastructure, and with the everything-is-a-model MDE principle, we implemented this correspondence as a weaving model called _DIASUITECOMP_REST_. The _DIASUITECOMP_REST_ weaving model contains two weaving links: the first link connects the Route resource of the RESTLANG model with the BusOnAirService component in the DiaSpecComp model (see Figure 5.22); the second link connects the Direction resource of the RESTLANG model also with the BusOnAirService component in the DiaSpecComp model. In this case study, the semantics of this kind of links is that the resource in the RESTLANG model is implemented by the component in the DiaSpecComp model it is linked to.

It is important to note that when creating the BusOnAir architecture framework, we also defined a correspondence rule called _DIASUITECOMP_REST_ governing this correspondence (see Section 5.2.4). The _DIASUITECOMP_REST_ rule checks if each resource defined in our RESTLANG model in the WebServices view is linked to at least a DiaSpecComp component in the DiaSpecComp model of the System view. This helps architects in keeping architecture views consistent since MEGAF can be used to automatically execute that rule and thus to automatically check if such a rule is actually violated or not.

## 5.5 Reflection

In this chapter I showed how MEGAF can be used to (i) define a project- or system-specific architecture framework in order to precisely focus on the specific peculiarities and technological aspects of a given system, and (ii) use the architecture framework to define an architecture description conforming to it. Basically, each viewpoint defined in the framework has been instantiated into an architecture view, correspondences between views have been checked with respect to the correspondence rules defined in the framework, Furthermore, DUALLY has been used to automatically transform among different architecture models within the architecture description, and BYADL has been used to extend an architecture description language in accordance with the system-specific stakeholders concerns.

By recalling Figure 5.1, it is evident that the process suggested by the combined use of MEGAF, DUALLY and BYADL is composed of two different phases: the first one is about the creation of the architecture framework, the possible extension of the used ADLs, and the possible generation of model transformations between ADLs; the second phase consists in the definition of the architecture description. Clearly, the second phase is more light-weight and it builds on the various artifacts generated by the first phase. This is because the generic MEGAF engine running under the provided megamodel _masks the complexity_ of using the modeling techniques applied in the first phase.

For example, the process suggested by DUALLY is organized into two different phases: the first phase (that ends with the transformations generation) is executed only once for
each ADL and is managed by a subject who retains a deep knowledge of both MDE tech-
niques in general and the metamodels to weave; the second phase is executed every time
a software architect needs to pass from a source architecture model to the corresponding
target architecture model. So, DUALLY works at two abstraction levels, therefore mask-
ing to the software architects the model transformations technology. The role played by
the software architect is, in fact, that of a final user that has only to model the system
SA using an ADL and will automatically have the system SA in each ADL linked to
A_{0}. On the same line, also the process suggested by BYADL is organized into two different
phases: in the first one, metamodels are composed in order to define the abstract syntax
of a new ADL, then also concrete syntaxes and model migrators are generated. At this
point, in the second phase, all the artifacts generated in the first phase are managed by the
MEGAF generic engine without asking the direct architect’s intervention.

For what concerns MEGA, this case study demonstrates an example of framework that
cannot be found “on the shelf” but must be constructed. This demonstrates the need of
realizing a framework from available viewpoints. MEGA encompasses this problem by
allowing software architects to create their own framework by reusing artifacts previously
declared in MEGA (e.g., I reused the weaving models defined in previous works when
importing the UML_{cc} profile and the FSP metamodel). It is then evident that the more
MEGA is populated with architectural elements, the more effective reuse becomes. As
the MEGA repository grows larger, more advanced features for searching elements in
the repository will be warranted; this is part of future work (see Section 7.1).

It is important to note that in MEGA reuse is possible at different levels of granularity:
from the reuse of a framework to the reuse of a single system concern. More powerful
mechanisms for supporting reuse are under study. At the moment an artifact can be reused
as it is. Modifications can be made by hand, possibly starting from a copy of the existing
element. I am currently investigating automatic extension and customization mechanisms
by following the work I did for BYADL, which, by means of composition operators,
enables the customization and extension of ADLs. Note that extensibility in this setting
is not simple. Even the semantics of extensibility is not completely clear for all kinds of
architecture elements. Indeed, this is a very special case of extension that can be supported
by means of suitable extension mechanisms. In other words, as future work, I would like
to provide these fine-grain mechanisms that can be used by software architects to realize
their own extensions.

By focussing on how we used MEGA for creating the BOAAF architecture framework,
it emerges that BOAAF is specifically developed for defining the architecture description
of systems which are similar to BusOnAir. This use of MEGA is somewhat close to
what architects usually refer to in the field of software product lines (SPL). A sharp
definition of SPL is given by the Carnegie Mellon Software Engineering Institute and
says that an SPL is "a set of software-intensive systems that share a common, managed
set of features satisfying the specific needs of a particular market segment or mission and
that are developed from a common set of core assets in a prescribed way". Under this

\[ http://www.sei.cmu.edu/productlines/ \]
5.5 Reflection

perspective, if we consider the BusOnAir as an individual within a specific family $S$ of software systems, the BOAAF framework we defined via MEGAF can be seen as a kind of solution for defining the software architecture of systems within $S$. However, in this specific context MEGAF focusses more on the architectural aspect of the family of systems (that is, on their typical stakeholders, their concerns, and viewpoints), whereas an SPL-based approach keeps a fixed architecture (i.e., the reference architecture of the software family) and focusses more on the product being developed, on its specific features, and on their configuration in order to assemble the product. In any case, both MEGAF and SPL-based approaches share the principle of software artifacts reuse; however, on the one hand in MEGAF reuse is considered as being opportunistic, i.e., architectural elements like concerns definitions, viewpoints, etc. are all put into the MEGAF repository, opening for opportunities for their future reuse. On the other hand, SPL-based approaches are built on the concept of predictive reuse, that is, a software artifact resides in an SPL only when its reuse can be predicted in one or more products in a well defined product line (if there is no means to reuse a software artifacts, then it is not part of the SPL) 7.

Another aspect that requires further investigation is that an extension of the framework could create problems on architecture descriptions realized following the framework. This aspect recalls “co-evolution” work that aims to modify models in order to keep the conformity to their metamodels as those metamodels change [22]. This is possible since a megamodel defined within MEGAF is itself a model. This implies that a megamodel can be manipulated through standard MDE techniques, like model transformation, composition, model weaving, merging, co-evolution, and so on.

Software architects using MEGAF are supported by strong and powerful consistency checks that, as described in Section 4.3, are defined at varying levels of granularity:

1. conformance with respect to the ISO/IEC/IEEE 42010 standard and rules defined therein,
2. checks of rules that are architecture framework-specific,
3. checks of rules that are “local” to the specific architecture description.

While the rules at point 1 should be defined once and forever, rules at point 2 allow software architects to properly define or constrain a framework, and rules at point 3 allow software architects to properly realize the software architecture description of the system-of-interest.

Furthermore, MEGAF enables the possibility to compare frameworks, such as to detect overlaps between two existing frameworks. Specifically, an overlap could occur at the conceptual level; such as two frameworks addressing the same concerns (e.g. performance) or in the model kinds used to frame those concerns. By comparing their respective metamodels, one could determine whether the two frameworks are treating performance

7http://www.methodsandtools.com/archive/archive.php?id=45
in the same way, overlapping ways, or different ways, based on reconciling elements of their metamodels to whatever necessary degree. The browsing tools of MEGAFA can assist this and there may be future possibilities for automated support in such comparisons.

From a practical point of view, weaving models and the realization of the model weaving operation might be an important bottleneck for the applicability of the proposed approach in industry. This issue can be rooted to (i) the user interface provided by the tools which is still rough, and (ii) the concept of weaving model itself, which may be of difficult comprehension for practicing architects since in general they are not fully trained on MDE concepts. To this purpose, I am evaluating the possibility to make model weaving more sustainable, thus promoting it into a more cost-effective solution to be adopted in production. In this context, usability and facilitating the creation of weaving models for (i) customizing and extending ADLs with BYADL, (ii) creating correspondence rules with DUALLY, and (iii) creating the various correspondences that may exist between architecture views with a low learning curve are fundamental for industrial adoption. It is important to note that this issue does not impact the whole approach; indeed, the first two kinds of weaving models (i.e., the ones pertaining to BYADL and DUALLY, respectively) are not directly created by practicing architects since it is up to MDE experts to defined those kinds of weaving model. Software architects will only interact with the various artifacts generated by the MDE experts (such as model-to-model transformations for DUALLY, textual and graphical editors for BYADL, etc.). So, basically weaving models are directly manipulated by practicing architects only when they are creating the various correspondences among the models defined in an architecture description. Under this perspective, future work will focus on two main directions:

- providing a means to automatically obtain correspondences among the architecture views defined within an architecture description. In so doing, I will relieve software architects of the burden of creating those correspondences (which may be a long and error-prone activity), demanding their creation to the underlying tool, and thus making the overall architecting process less mechanical and error-prone;

- enhancing the user interface of each part of the proposed approach. I will work on the usability of the approach, asking to the architects less interaction as possible with the underlying MDE tools (e.g., model transformations, model weaving, etc.). Ideally, MEGAFA will mask completely its underlying MDE mechanisms, providing a slick and usable tool support to software architects, which may be even unaware of the MDE techniques underlying the MEGAFA, DUALLY, and BYADL approaches.

Future work will address also the definition of scenarios in which practicing architects really need adaptation (e.g., when correspondence rules are fundamental within an architecture framework); in this context, I will elaborate on a methodology for (i) evaluating when and how the proposed approach may be applied depending on the involved risks, and (ii) mitigating those risks, when possible.

It is interesting to note that the last two correspondence rules defined for the BusOnAir framework (i.e., the UML_FSP and DIASUITECOMP_UML rules in Section 5.2.4)
operate on models that have been generated by DUALLY. More specifically, the FSP model of the UML_FSP rule has been generated by DUALLY by transforming a UML model firstly to $A_0$ and then to FSP (see Section 5.4.2); similarly, the UML model of the DIASUITECOMP_UML rule has been generated by DUALLY by transforming a DiaSpecComp model firstly to $A_0$ and then to UML (this part of the case study has been described in Section 5.4.1). Basically, the last two correspondence rules have been defined because at the beginning of the project we did not know that some of the involved models were related by DUALLY, so those correspondence rules were necessary. However, if we knew from the beginning that those models were the input and output of some model transformation process, those correspondence rules were unnecessary because the correspondences between those architecture models exist by construction thanks to the DUALLY transformations. So, in the future I am planning to better investigate on the relationship between the model transformations generated in DUALLY and correspondence rules in MEGAF. As a quick example, when the architect is executing a model transformation generated by DUALLY, MEGAF may automatically create a correspondence between the source and target architecture models. In so doing, the generated correspondence can be considered also as a trace between the involved models and can be governed by correspondence rules, without forcing the architect to manually define correspondences between architecture models.

For what concerns DUALLY, I believe that one of its most interesting points is the star topology in which architecture description languages are arranged, with $A_0$ as center of the star. In order to better understand the advantages and disadvantages of having this star topology of languages, I identified a set of generic dimensions to reason about which strategy may better suite an architect’s needs:

1. number of required transformations (NTrans) for achieving interoperability among $n$ ADLs,
2. number of additional transformations (AddTrans) required when a new ADL is added,
3. accuracy (Acc) of the transformations results.

Other secondary dimensions are also considered in the remainder of this section. Table 5.1 summarizes the main findings.

NTrans. From the first row, we can see that the point-to-point strategy requires the development of order of $n^2$ transformations, while the pivot-based solution requires order of $n$ transformations. Since many ADLs exist in the field of software architecture, the traditional point-to-point strategy would require possibly too many transformations. Specifically, since the long-term intent of DUALLY is to store and link several additional ADLs, the pivot metamodel proposed in DUALLY seems to be more appropriate.

AddTrans. Moreover, the second row points out that adding a further ADL to the interoperating network of ADLs requires $n$ new transformations implemented from scratch in
Table 5.1: Comparison between point-to-point and DUALLY

<table>
<thead>
<tr>
<th></th>
<th>point-to-point transformations</th>
<th>DUALLY</th>
</tr>
</thead>
<tbody>
<tr>
<td>NTrans</td>
<td>(n^*(n-1))</td>
<td>(n)</td>
</tr>
<tr>
<td>AddTrans</td>
<td>(n)</td>
<td>2</td>
</tr>
<tr>
<td>Acc</td>
<td>semantic loss</td>
<td>semantic loss + pivot inaccuracy</td>
</tr>
<tr>
<td>automation</td>
<td>manually coded transformation</td>
<td>automatically generated transformations</td>
</tr>
<tr>
<td>steps</td>
<td>one-step transformation</td>
<td>two-step transformation</td>
</tr>
</tbody>
</table>

the point-to-point strategy. The pivot metamodel strategy in DUALLY, instead, requires only one new transformation between the added notation and \(A_0\). More importantly, the full-mesh strategy requires a deep knowledge of the \(n\) existing notations, while the pivot-based solution requires knowledge on \(A_0\) and the ADL being added only.

Acc. As outlined in the third row, loss of information may happen in both the point-to-point and pivot-based strategies, since the expressiveness of the ADLs to relate can vary (i.e., if a language is more expressive than the other, the extra expressiveness cannot be typically mapped). Apart from this, the pivot-based strategy is in principle less accurate than the point-to-point solution, since the (domain-specific, but generic) intermediate model can increase the probability to loose concepts during the translation. This is a general problem when having a pivot language; indeed, even a well designed pivot metamodel might be less accurate than a point-to-point transformation. Heterogeneity between language to relate exacerbates this issue. In order to limit such an issue, DUALLY provides extensibility mechanisms as a way to minimize the loss of information. As shown in Section 5.2.3, we extended the base \(A_0\) metamodel with behavioural information. Specifically, we added the concepts of State, Transition, guard, etc. This allowed us to not lose behavioural information while passing through \(A_0\); this was an undesirable issue because elements would have been lost only because of the pivot metamodel and not for a real mismatch between the interoperating languages. As illustrated in Section 5.2.3, this activity has not required a big effort, due to the similar conceptual elements exposed by the ADLs being considered and the similarity between the concepts already in \(A_0\) and those in the ADLs. Indeed, when deciding to use DUALLY to relate languages in a completely different domain, extending \(A_0\) might not be enough. I am currently evaluating how to deal with such a situation, and which parts of DUALLY could still be exploited for models interchange.

The fourth row points out that while traditional point-to-point transformations require ad-hoc and manually coded transformations in some transformation language (e.g., ATL \[84\] or QVT \[115\]), DUALLY automatically generates the transformation code out of the weaving models.

The fifth row puts in evidence that DUALLY uses a two-steps transformation, while a single-step is required in point-to-point strategies. The main issue about the two strategies is that of accuracy, i.e., passing through a pivot metamodel could lower the quality of the models produced by the transformations; in this case study we managed to overcome this
problem extending $A_0$ so that no common element is lost because of $A_0$’s expressivity.
So, in this specific case, extending $A_0$ and applying the \textsc{Dually} approach leads to a comparable degree of accuracy as if a point-to-point strategy is applied. In addition, applying the pivot-based strategy opens for interesting future works; for example it is possible to reuse the weaving models defined for linking UML\textsubscript{cc}, DiaSpecComp, and FSP to $A_0$ while relating other languages to $A_0$, or quantifying how many elements UML\textsubscript{cc}, DiaSpecComp, and FSP have in common with respect to other languages being added to $A_0$.

For what concerns \textsc{ByADL}, in the BusOnAir case study we used it in two situations:

- to extend the domain- and computing paradigm- specific DiaSpec language in order to being able to model the whole BusOnAir system, rather than only the Bus System.

- the (automatically generated) \textit{DiaSpec2DiaSpecComp} model migrator has been used to obtain an initial version of the DiaSpecComp model starting from a base DiaSpec model. Such initial model then has been considered as starting point to model the whole BusOnAir system.

Under this perspective, it is important to highlight the fact that DiaSpec is a textual-only notation and that we managed to obtain a \textbf{graphical notation} for DiaSpec simply by extending its metamodel with \textsc{ByADL}. From the architect’s point of view, that is, when we were creating the architecture description (see Section 5.4.1), we obtained a graphical notation for DiaSpec with no effort at all. Indeed, it has been up to the \textsc{ByADL} generic composition engine to lookup for an available graphical representation for the concepts in DiaSpec and to assign it to the corresponding concepts in the composed DiaSpecComp metamodel. More specifically, the graphical representation for each DiaSpec concept has been assigned by (i) automatically navigating the semantic link between the DiaSpec element and its corresponding $A_0$ element (which is annotated with information about its graphical representation), and then adapting the annotation of the $A_0$ element to its associated element in the resulting DiaSpecComp metamodel.

Moreover, extensions in \textsc{ByADL} are defined in an \textbf{ADL-independent} manner and they are stored in the \textsc{MegaF} megamodel, so to be reused for further extensions of (even different) ADLs. It is important to highlight that the approach promoted by \textsc{ByADL} is incremental, so that software architects are able to extend and customize their ADL whenever required. This is particularly important in practice since often the characteristics of “optimal” ADL (for instance as required by the considered non-functional aspects) may change during the architecting phase [120].

However, the current version of the \textsc{ByADL} approach suffers of some limitations, I will discuss them in the remainder of this section. Firstly, the \textbf{semantics} of the extended ADL is defined by means of relationships with $A_0$. Therefore, a precise semantics cannot be
expressed for elements that are not in $A_0$. For instance $A_0$ does not explicitly handle hardware devices. Consequently hardware components should be typed as generic SA components. However, a possible help for alleviating this limitation come from DUALLY’s extensibility; indeed $A_0$ has been defined as extendible and DUALLY provides dedicated extension mechanisms for it [37]. In this way $A_0$ can be extended with domain-specific concerns, thus enabling a more precise semantics for the considered domain (i.e., $A_0$ can be extended with hardware aspects and with explicit hardware components). I am planning to investigate more powerful and formal ways to provide semantics to the ADLs that are extended or customized in BYADL.

Focussing on the composition operators provided by BYADL (see Section 3.3.1), there is no evidence that the defined operators are enough for extending any existing ADL. New operators can be incrementally added within the BYADL framework in a modularized way. Technically, adding a new operator means to (i) extend the Operator metaclass in Composition Metamodel, (ii) update the Composition2Ecore transformation with the execution logic of the added operator and (iii) update the BYADL HOTs in order to generate migrators reflecting also the logic of the added operator.

As said before, the generation of graphical editors is in a prototypical stage and it is not able to automatically solve every possible conflict that may arise. This aspect needs further investigation. An important aspect that should be considered is that software architects associate to graphical elements a semantics. When inheriting a graphical element we inherit also this implicit semantics that must conform to the one associated to the element by the tool.
Software architecture and model-driven engineering are the two pillars of this dissertation. Indeed, even if the main focus of this dissertation is on software architecture, it has been natural to also deal with pure model-driven engineering techniques and to provide contribution also in this research field. In the following, I will describe how the research described in this dissertation relates to other work in these research fields (i.e., SA and MDE).

6.1 SA-SPECIFIC APPROACHES

For what concerns of ADLs interoperability, in [61] the authors propose an approach based on consistency rules, and bidirectional model transformations to automate the synchronizations of AUTOSAR (Automotive Open System ARchitecture)\(^1\) and SysML (System Modeling Language) models\(^2\). Even though the approach is general and can be applied on any couple of modeling languages, it differs from DULLy since the used model transformations which underpin the synchronization mechanism are manually written and are not organized in an extension hierarchy as proposed in this paper.

The Acme initiative [59] is famed for being one of the very first technologies to tackle the problem of architectural data interchange. Acme and its supporting tool ACMESudio were born as a simple, multi-style modeling environment also providing the possibility of using it as a common interchange platform for multiple ADLs. It eventually drifted away from this particular field now standing on its own as an architectural description and analysis environment. Acme also provided foundations and mechanisms to extend both its core language and tool. Tooling extension points come from allowing other tools to physically read and write Acme descriptions while semantic extension is enforced by allowing properties to carry ADL-specific data within the model. The provided ACMElib library can also be used to adapt ADL technologies to Acme and to allow their manipulation within ACMESudio. Some additional efforts brought Acme closer to UML by providing a ready-made profile for this task [65]. A number of reasons can be presented

\(^1\)http://www.autosar.org
\(^2\)http://www.sysml.org
for Acme drifting somewhat away from its initial goal, the main of which is that its core technology does not provide direct support to integration with other architectural description frameworks. More specifically, Acme comes with libraries for parsing, un-parsing and manipulating the representations that aim to help software architects to integrate new ADL descriptions. However the hard work in relating two different ADLs is not properly supported, as discussed below. Contrarily to DUALLY, software architects are obliged in Acme to explicitly write end-to-end transformations among each pair of ADLs. As described in [58], in order to integrate two ADLs, e.g., Wright and Rapide, the Acme language needs to be augmented (through annotations) with specific information coming from Wright, and peculiar to it. Successively, the augmented model needs to be extended even further, with Rapide specific information (and therefore, yet again the kernel might itself need augmentation). As claimed in [58], the hard work occurs in the middle step, i.e., when bridging the semantic gap between Wright and Rapide, and this step is not properly supported. This brings no evident advantage in having an intermediate notation. It is also not evident how the annotation mechanism of Acme will work when considering ADLs very different from Acme. Also contrarily to DUALLY, Acme authors make clear in [58] that the true goal of Acme consists in simplifying the migration of one technology to Acme (and once there, utilize the Acme analysis and modeling technology present) by acknowledging the clear difficulty to relate two different ADLs utilizing the Acme approach. Furthermore, Acme does not provide support for the loss in translation and with the used technology is very difficult to argue about the correctness of the transformations. Moreover, efforts invested in integrating technologies within Acme are not reusable outside of ACMESStudio [59]. Descriptions instantiated within Acme will still have problems in “exiting” the technology and will depend heavily on its intrinsic format. DUALLY keeps the idea of a minimalistic kernel of elements acting as doorway for information migration. By means of fully automated transformations, DUALLY will allow for architectural information to be maintained and upheld. From an interchange specific perspective, DUALLY renders it simple to integrate a technology while keeping it core-independent, thanks to the provided transformation engine. Summarizing, the key differences are: (i) DUALLY’s A₀ is frozen in its essential core and provides attaching points within its metamodel so that (independent) extensions may be developed and included in the A₀ language; (ii) linking ADLs within our technology is an agile and architect-friendly modeling step, transformations are automatically generated by DUALLY once the process is complete, and all the modelled transformations and the relative weaving models are fully reusable in a model-driven fashion; (iii) the key concept of DUALLY is to allow interchange: the use of other technologies is welcome and the proposed approach acts as a bridge in order to reach other technologies, DUALLY does not provide any analysis techniques or technologies, it is out of the scope of this dissertation; rather, it can be considered as an approach to allow migration from one ADL into another.

xADL [33] is an active research effort born and in progress at ISR - University of California, Irvine. Just like DUALLY, the technology bases itself around a core of elements as a reference: xArch. xADL inherits from xArch a number of like-to-have features such as direct run-time instantiation of SAs, model grouping, SA hierarchy and so on. xADL as well as its core xArch, are based on XML and thus fully extensible [31]. The xADL
6.1 SA-specific Approaches

...project evolved into a powerful technology backed by a solid workbench, ArchStudio 4, which maximizes the possibilities the technology shows. The possibilities and limitations exposed by xADL on its own are very similar to those evidenced within Acme. Integration efforts have still to physically augment the common XML architectural description provided within xArch: in this sense, a typical scenario would imply extending the core once for one technology and yet again for another one. The same core will become bigger with the first extension and yet even bigger because of the second one: this will still make the extensions ‘depending’ upon one and the other. Following these points, a natural comparison is possible between xADL and DALLY: it is close to the strong-points and novelties introduced by xADL, and it also provides full compatibility and support to the MDA process. Furthermore, DALLY and xADL share a modular structure designed to be extensible. However, the main goal of DALLY differs from the one of xADL: xADL may be used to define DSLs, (i.e., structured languages engineered to be extended and ‘configured’ so as to represent desired domain-specific architectures), whereas DALLY is an approach for interoperability between ADLs and/or UML profiles. For what concerns BYADL, the use of XML schemas in xADL do not provide facilities to define the semantics of individual elements but only the syntax. The semantics is fundamental in order to define a common and well defined means to interpret the syntactic concepts. In xADL the semantics can be encoded into comments in the XML schemas themselves, in a project documentation, in visualization tools that stakeholders use or, finally, in analysis tools associated to a given feature [108].

UML 2.0 [116] has been used for modeling architectures and different profiles have been proposed. Even thought UML is a rich language composed of 13 different diagrams, they are not a ultimate solution for modeling every possible architectural concern. While extension mechanisms of UML allow software architect to define stereotypes and tagged values, which can be used to extend UML elements to better capture domain specificities, these mechanisms cannot be used to define new diagrams. These extensions cannot fully represent all concepts of every ADL and on the other side, as already claimed in [110, 108], it is impractical to have a “universal” notation.

AADL [48] was born as an avionics focused DSL and later on moved to represent and support embedded real-time systems: its name and acronym were then re-factored to Architecture and Analysis Description Language. Standardized by the SAE (Society of Automotive Engineers) AADL now shows impressive usage in the field of both software and hardware specification and validation. AADL was designed as an extensible core language supporting modeling from multiple aspects and points of view. The validation technology it provides, addresses timing and performance properties of embedded real-time systems. Concurrency and interaction semantic specification is enforced. AADL was also one of the very first ADLs that introduced the concept of ‘feature’ as a component’s way to interact with others at the implementation level. The extension mechanisms of AADL include the definition of custom properties to specify additional ADL-specific analyses and/or generic information to be attached on the architectural design. Additional notation extension efforts are bringing AADL closer to UML-friendly notations (via a profile) [49]: the initiative has developed a UML profile to later on synchronize the two
technologies. In addition, the main open-source tool of the technology, OSATE tool, is very close to our view of an ideally extensible framework: it supports plug-ins, core-set extensions and might be made to support MDA specific technologies, by coordinating it with the Eclipse MDA initiative. Other extensibility mechanisms defined through constructs such as the standard “annex” plus the mentioned property set extensions are closely related to our view of “semantic” enhancement, which is considered very important in any technology. My research effort took AADL as a key reference due to the exposed features and also because of its widespread adoption within dependability critical industrial development processes. Unfortunately, AADL does not provide automated support to its extensibility possibilities. Somewhat like xADL, AADL can be viewed as a modeling notation that can be complemented by description technologies tailored to specific goals of a particular modeling view and it can be considered a language supporting DSL generation. We decided to follow other recent trends and research directions in the ADL field confirming that interoperability between these domain-specific realities is still to be realized effectively. As a consequence, the goal of DUALLY is that of providing automated interoperability and synchronization mechanisms between technologies.

ADLs in DUALLY are organized in a star-topology centered around a core set of essential descriptive elements: $A_0$. This idea derives primarily from the requirement of providing a common minimal ground language to enable all architectural description languages to “communicate” between one another. This communication is a mid-way layer between every integrated technology. Previous technologies and some UML-based attempts at integration or architectural description, underlined the need for a basic set of meta-elements to be used as a reference set. A number of different yet close attempts both at the description and formalization of such a set may be found in [81] and [1]. [81] presents an interesting example of meta-information exchange between ADL formats via model transformation. This particular case shows a glimpse of the full potential that can be achieved within DUALLY, its transformations and our transformation engine. Considered technologies are Acme and META-H, two of the previously mentioned architectural modeling technologies. The paper shows in detail the actual implementation of the meta-interchange to take place from Acme to META-H, as it is carried out within the Eclipse framework. The same principles are applied within DUALLY with the chief difference that the shown transformations will execute mainly on model instances whereas DUALLY is able to encompass a much wider scope. In fact, DUALLY’s interchange engine will make sure that interoperability is provided at both modeling and metamodelling levels. In [1], Smeda et al. bring around a relatively new concept in the field of architectural description: that of meta-architecture description language, i.e., Meta-ADL (MADL). The paper focuses on the specification of a meta-level to describe ADLs right from their core level, rather than architectures themselves. The paper shows the undoubted potential behind the specification of this meta-metamodel for architectural description languages, and provides a possible implementation of such a technology. The approach however shows some shortcomings: the development of a totally stand-alone architectural “MOF” is both expensive in terms of effort and compatibility with industrial standards (e.g., UML). Utilizing For example, such a technology would mean to sacri-

3http://www.htc.honeywell.com/metah/prodinfo.html
vice compatibility with UML right from its MOF. I believe that this incompatibility is unacceptable in the modern architecture description research field. [1] also provides a detailed assessment of architectural description issues that are yet to be solved (such as full standardization, architectural comparison, integration and interchange of formats, styles, etc.). While MADL effectively tackles these issues, some newly opened ones remain unhindered: industrial integration, technology roll-out, knowledge re-use and a number of other issues remain untouched. DUALLY tries to work on the same scope as in [1] while also trying to “think industry” by considering the industrial prospects and problems on architectural description. Indeed, DUALLY (together with BYADL and MEGAFL) and is integrated with a mainstream IDE such as the Eclipse modeling and development framework as well as providing full support to model transformation, metamodel transformations and a number of MDE techniques and technologies; all the approaches proposed in this dissertation also work closely with the UML and MOF (EMOF), de-facto standard in modeling notations worldwide.

In [41] the XTEAM framework has been proposed for creating Domain Specific Development Infrastructures (DSDIs). XTEAM is a model-driven engineering (MDE) approach to software architecture that combines extensible modeling languages based on architectural constructs with a model interpreter framework that enables rapid implementation of customized dynamic analyses at the architectural level. In XTEAM, metamodel composition is the initial step for creating modeling infrastructures. Metamodels are linked to a proposed metamodeling language (called ACT) in order to establish their semantics. The approach has been implemented within the GME environment which gives the possibility to have an unrestricted number of levels in the modeling architecture. In fact, ACT is defined as a metamodeling language in terms of another metamodeling language. This aspect locks the specified ACT models with GME, since other technologies like EMF or MOF do not give the possibility to define other metamodeling languages prior the definition of metamodels. XTEAM promotes domain-specific analysis through the use of extensible Model Interpreter Frameworks (MIFs) that transform the composed metamodel to analysis-specific notations. However this is done programmatically and not through a dedicated architectural interoperability framework like in DUALLY.

In general, all the above mentioned attempts to create extensible and domain- and style-specific ADLs do not sufficiently take into account the problem of reusing already defined extensions. In particular each extension is coupled and specifically defined for the particular ADL and cannot be reused for extending other ADLs. Moreover, these approaches propose extension mechanisms that only partially satisfy requirements modern software architects. For instance, as described before, Acme supports the extension of its semantics only by means of properties that cannot be easily interpreted by tools. The lack of semantics can lead to misunderstanding and specifically, as described in [41], the lack of semantics within metamodeling languages can lead also to several practical problems: i) imprecise determination of semantic relationships between the used ADL and the elements that must be integrated within the ADL, ii) onerous manual composition of notations, and iii) lack of rigorous and automated validation mechanisms. Moreover, ADLs

4GME - Generic Modeling Environment: http://www.isis.vanderbilt.edu/projects/gme
interoperability is not addressed properly by any language; Acme provides some initial ideas and more mature results can be found in the DUALLY approach.

A discussion of existing architectural frameworks, together with their limitations and challenges, is provided in Section 2.1.7. Most architecture frameworks in use are closed (i.e., they cannot be easily adapted to new needs), and the construction of existing ones requires a complete rework since reuse is not supported at all. Tool support for architecture frameworks exhibits this limitation as well: automated support, where it exists, follows the predefined viewpoints and models; support for developing extension of architecture frameworks in terms of new viewpoints is non-existent.

Some contemporary ADLs allow for the possibility of their extension to support new viewpoints. Two OMG projects: SysML\(^5\) and UPDM (Unified Profile for DoDADF/-MoDADF)\(^6\), both built upon UML, provide stereotypes for Stakeholder, Concern, View and Viewpoint, following the IEEE 1471 standard, but do not require the rules of that standard be followed in applying these stereotypes.

Radjenovic and Paige describe AIM (Architectural Information Modeling) as a generic, extensible platform for modeling systems and software architectures [127]. Unlike the present work, AIM maintains a single, central reference metamodel from which multiple views are projected.

### 6.2 Generic MDE Approaches

For what concerns models interoperability, related work mainly regards model-based tool integration, automatic derivation of model transformations and semantic integration. The authors of [88] present ModelCVS, a framework similar to DUALLY in which metamodels are lifted to ontologies and semantic links are defined between such ontologies, they will serve as a basis for the generation of model transformations; this approach manages also concurrent modeling through a CVS versioning tool. DUALLY is different because it implies the $A_0$-centered star topology (it scales more when dealing with multiple notations) and the preliminary step of metamodel lifting is not performed.

Furthermore, the role of $A_0$ in DUALLY is similar to the Klaper language in the field of performability (i.e., the combined measures of performance and reliability). Grassi et al. in [66] propose the Klaper modeling language as a pivot metamodel within a star topology; the Klaper-based methodology is different from DUALLY approach since model transformations are vertical (whereas in DUALLY they are “horizontal”) since Klaper is designed as a means between design-level and analysis-oriented notations. Moreover model transformations are not derived from semantic bindings (i.e., weaving models in DUALLY), they must be manually developed.

---

\(^5\)http://www.sysml.org

\(^6\)http://www.updm.com
A number of existing approaches to models synchronization and change propagation is available today, imposing different restrictions on the underlying transformations. In general it is required that there is a one-to-one relationship between source and target changes. In point of fact, model transformations are neither total, nor injective, thus posing a number of difficulties and requiring appropriate support for change propagation [70]. An attempt to support change propagation is presented in [143], where an imperative paradigm capable of backtracking, and a language used to handle declarative aspects are used to build an hybrid method trying to deal with tool integration mechanisms. A declarative evolution of the mentioned work is described in [144] where the author illustrates a technique to implement a change propagating transformation language called PMT. After the first mapping from source to target, manipulations can occur on both models. When a new transformation is required the target model is updated in a non-destructive manner by exploiting the trace information generated from the previous execution. In this respect, PMT can be considered as an alternative implementation of the DUALLY model transformation engine that takes into account (and persists) target modifications when new transformations are performed. The combination of bidirectionality and change propagation can enable the support of model synchronization. In general, the existing approaches impose relevant restrictions on the characteristics of the involved transformations. For instance, approaches like [147, 62, 114, 53] require the mappings to be total, while [147, 62, 150] impose the existence of some kind of bijection between the involved source and target. In particular, [150] proposes to automate model synchronization from model transformations. It is based on QVT Relations and supports concurrent modifications on both source and target models.

Over the last years a number of work has been proposed to cope with the problem of tool integration and interoperability in MDE. Such works can be classified into Transformation-based approaches and Metamodel integration approaches [136]. The former approaches, like [43, 140], propose the adoption of model transformations which aim to serve as a bridge between the various tools that have to interoperate. In particular, model transformations are used to transform data required by heterogeneous tools. Differently to DUALLY, such approaches rely on manually written transformations defined with respect to the notations adopted by the considered tools. Metamodel integration approaches, like [6], rely on the definition of a common metamodel to establish tool interoperability. Even though such approaches are similar to DUALLY, they do not provide mechanisms supporting the extension of the common metamodel.

The problem of interoperability has been tackled also in the context of model-to-model transformation languages. In [85] the authors propose an approach based on a Common Intermediate Language to support interoperability between different model transformation languages. Differently from our approach the authors analyze a set of well-known transformation languages and identify common characteristics which are captured in a common metamodel which is not extensible.

Going back to the nineties, a family of works have been proposed to exploit a single formal kernel language to integrate specifications written in different languages. One of the most
prominent work in this family is the one by Jackson and Zave [152] in which Z is used as a
common semantic domain for the composition of partial specifications defined in different
languages. The resulting composed specification is then used to check the consistency of
the initial partial specifications. Under this perspective, the goal of DUALLY is quite
different since it considers the A₀ hierarchy as an intermediate means for transforming
models across different languages, rather than a way to check their global consistency.

There is also some related work using megamodeling techniques, testifying that meg-
modeling can be successfully used in contexts in which models need to be navigated,
composed, managed and represented in different ways. The work presented in [55] ap-
plies megamodeling techniques to the model driven performance engineering process. In
this work the core metamegamodel has been extended with three concepts: annotation
model, trace model, and transformation chains. In our paper the situation is more com-
plex since our approach must be totally generic since it cannot focus on a specific domain
only and since we deal with cross-layer artifacts like the weaving models linking meta-
models. Other works on megamodeling have been developed, like the one in [104] that
proposes an approach to automatically build a usable cartography of existing software
platforms by merging generated metadata with user-specified metadata; in this work the
authors propose a solution based on a combination of megamodeling, model transforma-
tion and model injection. Even if our work shares some technological aspects with the one
proposed in [104] (i.e., megamodeling, model transformations for automation, etc.), the
scope and objectives of the two works clearly differ since our proposal is for architecture
descriptions, whereas they focus on a more technological problem.

For what concerns metamodels composition, the BYADL operators have been defined
building on those presented in [38] and [141]. In particular, the operators proposed in [38]
have been made applicable on Ecore metamodels and those introduced in [141] have been
refined in order to lighten the composition process. For example, the deletion of elements
is implicitly performed in BYADL by a slicing algorithm. Moreover, the BYADL ap-
proach is somehow parallel to that proposed by Ledeczi et al. [95]: in that approach the
structure of the metamodels to compose cannot change (so that original models are still
valid). BYADL migrators permit to relax this constraint, providing more flexibility while
designing and applying the composition operators. Aside from the composition operators,
the BYADL conflict resolution mechanism is based on the work presented in [124].
CHAPTER 7
CONCLUSIONS

This dissertation can be considered a contribution to the study of techniques to effectively engineer architecture frameworks and languages. More specifically, in this work I focus on Model-Driven Engineering techniques for supporting the definition of architecture frameworks with a special consideration for views consistency and architecture description languages extensibility.

To this purpose I propose an approach for defining, reusing, and sharing architectural elements across different projects (and within the software architectures community). With the term architectural element I mean all the concepts that are relevant during the architecting activity, such as:

- the stakeholders of the system,
- system stakeholder concerns,
- architecture description languages,
- architecture views and viewpoints,
- architecture models,
- correspondence rules integrating viewpoints.

As discussed in Section 3.1, at the core of the proposed approach is a model-driven infrastructure for describing the architecture of a software system by adapting existing architectural languages, views and architecture frameworks to domain- and organization-specific features. The proposed approach builds upon the conceptual foundations of ISO/IEC/IEEE 42010 for architecture description and it is totally independent from the used architectural elements. In this context, modern software architects can exploit the proposed approach as the basis for developing new architectural frameworks, that will be used to specify architecture descriptions and perform different kinds of analysis on them. Such an infrastructure has been implemented by means of a combination of model transformations, weaving models, and megamodeling techniques (see Section 4.3 for the details on the implementation of the proposed approach). This provides the needed level
of automation that an acceptable approach in the field of software architecture must have. For example, automatic consistency checking mechanisms have been defined in order to check if an architecture description is actually conforming to the architecture framework governing it; in turn, an architecture framework can be automatically analysed in order to check if it conforms to the directives of the ISO/IEC/IEEE 42010 standard, and so on.

Architecture frameworks that have been built using the above described infrastructure can also benefit from a dedicated facility for solving one of the harder problems in the field of software architecture [47]: architecture views consistency. To this purpose, such a facility allows architects to automatically transform among architecture models, after having defined the relationships that exist between the used ADLs. Details about this part of proposal are provided in Section 4.4.

Moreover, similarly to what has been proposed for architecture frameworks definition, extension and reuse, I propose a dedicated approach for managing ADLs extensions and architecture concepts reuse across ADLs. Under this perspective, such approach allows architects to extend an ADL depending on the system’s stakeholder concerns. For example, an architect can extend an existing ADL with new concepts coming from domain-specific concerns, new architectural views, analysis needs, etc. It is important to note that ADL extensions are defined in an ADL-independent manner and they are collected in the MEGAF infrastructure so that they can be reused for future extensions of other ADLs. Details about this part of proposal are provided in Section 4.5.

The approach proposed in this dissertation has been applied on a real-world case study called BusOnAir. The project is about a generic information system for managing real-time information about public transportation systems and making it available as web services. BusOnAir is currently under development as an R&D project at the University of L’Aquila (Italy) and its launch is planned for mid-summer 2012.

In the following I will present the results achieved so far in this research work. Each result is defined in the context of the corresponding research questions described in Section 1.3.

RQ1 Is it possible to reuse and compose already existing viewpoints, languages, design decisions in order to shape an architecture framework that better fit with the system’s stakeholder concerns?

MEGAF is an infrastructure for realizing architecture frameworks with a focus on (i) reusing already defined viewpoints and languages and (ii) considering stakeholders and their concerns as part of the framework itself. MEGAF is built on the conceptual foundations provided in the ISO/IEC/IEEE 42010 standard so that its concepts can be easily understood and manipulated by software architects. MEGAF allows software architects to create an architecture framework and to specify the necessary views, viewpoints, stakeholders, concerns of the system that are part of the architecture framework itself. MEGAF provides a built-in mechanism for navigating among
the architectural elements defined in the framework and to check its conformance to the ISO/IEC/IEEE 42010 standard.

Correspondences and correspondence rules between arbitrary architectural elements, including stakeholders, system concerns, models, etc. can be created in MEGAF. More precisely, they enable the architect to express and enforce relations both between various elements inside an architecture description and across architecture descriptions.

Furthermore, MEGAF provides functionalities that allow software architects to create their own framework by properly selecting among artefacts previously defined and resident inside MEGAF. Once the framework has been defined, it can be used to realize the architecture description of the system-of-interest. In other words, MEGAF allows to create an architecture description conforming to the architecture framework, to specify views that are governed by viewpoints defined in the architecture framework and that address some previously-specified system concerns.

It also allows to check consistency among arbitrary architectural elements. MEGAF also supports different kinds of checks:

- conformance of an architecture description with ISO/IEC42010;
- compliance of an SA description to an architecture framework;
- satisfaction of framework-specific rules;
- satisfaction of rules depending only on the system-of-interest.

From a technical point of view, MEGAF has been realized as an extensible repository of viewpoints, views, model kinds, architecture models, system concerns, and stakeholders. Such a repository has been implemented as a megamodel (i.e., a model whose first-class elements are other models). The use of megamodelling promotes the reuse of any architectural element that resides in the MEGAF infrastructure because in MEGAF a framework can be created either by linking and reusing existing elements or adding new ones if needed. More details on the implementation of MEGAF are provided in Section 4.3, whereas its application to a real-world case study has been described in Section 5.2.

RQ2 Is there a way to automatically keep consistent architectural views (and the models belonging to each of them)? Is it possible to reuse their commonalities and let them interoperate? Further, is there a scalable way to add (or remove) an architectural language from the set of used languages within the architectural framework?

In MEGAF cross-view consistency is supported in two different ways. The first one is more descriptive, and allows architects to define a set of correspondences between architectural elements within an architecture description. In the architecture framework, those correspondences are navigated by either generic or framework-specific correspondence rules which check if the involved architectural elements are in a consistent state. MEGAF allows architects to define correspondence rules via OCL (i.e., a standard language in model-driven engineering for defining constraints), and thus to execute them on specific architecture descriptions. This aspect of MEGAF is
described in Section 3.1 and some examples of correspondence rules are provided in Section 5.2.4.

The second cross-view consistency mechanism is supported by the DULLLY interoperability engine. Basically, it allows architects to define proactive correspondence rules: when an inconsistency is detected between different architectural models, DULLLY actively transforms the involved models in order to restore their consistency. Conceptually, DULLLY is a transformation engine that permits to transform an architecture model $am_1$ into a corresponding architecture model $am_2$. Each $am_i$ conforms to its $ADL_i$. Such a transformation passes through $A_0$: a pivot metamodel specific for the SA domain. The main purpose of $A_0$ is to represent the commonalities of current ADLs. Whenever a model $am_i$ has to be transformed into $am_j$, a two-step process is performed by transforming $am_i$ into $m(A_0)$, and successively $m(A_0)$ into $am_j$. DULLLY allows the automatic generation of the needed model-to-model transformations. Thanks to the $A_0$-centred star architecture a linear relationship between the selected language and $A_0$ is created, thus reducing the number of transformations needed. Also, DULLLY is scalable since adding an ADL to the topology of notations is straightforward: the software architect must simply link the new ADL to $A_0$, without considering all the other notations in the topology. The same rationale holds for removing an ADL from the topology of notations. The model transformation engine is completely hidden to software architects, that can continue using the most convenient architecture description languages, and seamlessly interchange information among them.

More details about the descriptive cross-view consistency is provided in Section 3.1.1, and the details about the DULLLY interoperability engine are given in Sections 3.2 and 4.4.

RQ3 Focusing on a single architectural language, is it feasible to extend or adapt it in order to get it closer to either the domain, development process or stakeholder concerns of the system?

BYADL provides an incremental model-driven approach for extending or customizing existing ADLs in order to adapt them to the ever-varying system stakeholders concerns. In this context, the adapted ADL is built by means of semantic extension mechanisms. The main input of BYADL is represented by the metamodel of the ADL to be extended; it is contained into the MEGAF megamodel, together with other metamodels representing either domain-specific concepts, analysis-related constructs, and so on. The metamodel of the initially considered ADL is extended by applying specific metamodel composition operators (see Section 3.3.1 to read more about them). Moreover, since the BYADL approach is incremental, its composition mechanism can be used to easily and quickly update ADL already stored in the MEGAF megamodel after a system concern is evolved. The ADL could be also simply customized; this is done by composing it with ad-hoc metamodels representing only the customization aspects of the language (i.e., only the entities that change or must be added).

The ADL obtained at the end of the process consists of (i) abstract syntax, i.e., the
7.1 Future Research Directions

On the ongoing and future research directions side, several activities can be carried on. First of all, as next steps I am planning to investigate on and address the limitations and issues highlighted in Section 5.5.

In the recent survey on the architecting activities in industry [105] it clearly emerges that nowadays the majority of architects describes the software architecture of a system in terms of multiple views, each view addressing a given set of system concerns (which in turn are held by different system stakeholders). It implies that special consideration in MEGAF should be paid to the management of multiple views, their consistency checks and on providing facilities to seamlessly consider the architecting activity from this perspective. In the light of these considerations, a future research work is to enhance the MEGAF approach so that it provides different solutions and system-views, depending on the current stakeholder using it. MEGAF will also filter system information (and thus provide different views of the architecture) depending on the current stakeholder. More specifically, MEGAF will provide only the information related to the concerns held by the current stakeholder, while hiding other aspects of the system depending on the stakeholder’s access rights; this results in a much more cognitively manageable and well organized architecture description of the system.

Elaborating more in this line, thinking about MEGAF (together with DUALLY and BYADL) as a distributed application can lead to a future collaborative environment for architecting. Indeed, different stakeholders may access MEGAF in a distributed fashion and may
share the various architectural elements (e.g., architecture models, viewpoints, design decisions) among them. This will (hopefully) lead to a more effective team collaboration, in which each team member "sees" only the portion of architecture description pertaining to him, and can communicate and share architectural elements with other team member in a more straightforward manner. Currently, I am working on this future research direction by providing a new model transformations approach in which a model transformation can be defined and used as a service. By doing this, I will provide a dedicated model transformations broker in which all the transformation generated by DUALLY and the model migrators generated in BYADL will be available to every architect. In this particular architecture, the role of the model transformation broker is that of the classical Service Broker in Service Oriented architecture; four are the main involved roles: transformation provider, Transformation engine provider, transformation broker, and transformation requester.

- **Transformation provider:** generates a model transformation by using DUALLY or BYADL, creates a basic description of the transformation (including also its source and target languages and where they are stored within the distributed environment), and publishes it to the transformation broker.

- **Transformation engine provider:** runs a model transformation execution engine (e.g., the ATL virtual machine), and makes it available to the transformation broker.

- **Transformation broker:** hosts a registry of model transformations and their descriptions; it is responsible for linking a transformation requester to a transformation provider. When it receives a transformation request, it selects the model transformation to be executed, selects the transformation engine provider whose engine matches with the language of the selected transformation, and asks the transformation engine provider to execute it; then, it gets the output models produced by the engine provider and forwards them to the transformation requester.

- **Transformation requester:** discovers an available model transformation by searching through the service descriptions provided by the model transformation broker, and asks it to execute the discovered transformation.

As anticipated in Section 5.5, as the MÉGAF repository grows larger, more advanced features for searching and visualizing elements in the MÉGAF megamodel will be needed. If on one side this issue may be partially addressed by the previously described multiple views management work, on the other side the standard AM3 user interface for visualizing megamodel is not enough (see Figure 4.3). More specifically, the standard AM3 user interface comes handy for having a quick view of the structure of the MÉGAF megamodel, but it does not show at all what are the relationships between the various architectural elements in it. Those relationships are fundamental while architecting a complex system; for example, think about quickly seeing which concerns are framed by a given viewpoint, which views must still be defines for a specific architecture description, or checking which elements in an architecture model correspond to elements in another architecture model,
and so on. To this purpose I will make a study about the most convenient way to show relationships between architectural elements, and I will realize it as an extension of MEGAF. As an example, Figure 7.1 shows how this representation may be realized in practice by building on a visualization kit in Eclipse called Zest.

The interesting aspect of the Zest visualization kit is that it has been designed to be fully extensible and easy to program against; it means that I can use it as starting point and then I will extend it with software architecture-specific facilities in order to provide an effective and pleasant user experience to the software architect. Furthermore, Zest views conform to the same standards and conventions as existing Eclipse views. Thus, the providers, actions and listeners used within existing Eclipse-based applications can be reused within Zest; it avoids me the effort to refactor or re-implement the code of MEGAF, DUALLY and BYADL.

Regarding interoperability among ADLs, I am also investigating on how to fully automate it. That is, currently in DUALLY a set of semantic links must be established between each ADL and the $A_0$ pivot metamodel. I am evaluating the feasibility to automatically generate that set of semantic links; this issue may be tackled by considering some heuristics or ontology-based approaches. Another important aspect is that of combining the DUALLY infrastructure with synchronization and change propagation approaches, so that only model changes are "exchanged" among the various ADLs.

As discussed in Section 3.2, the DUALLY star-topology of ADLs ensures the convergence of the model transformation process under the assumption that concurrent modifications to different architecture models cannot apply. An interesting work that will be carried on DUALLY is the study on how concurrent modifications can be managed in

---

2http://www.eclipse.org/gef/zest/
its star-topology of architecture description languages. More specifically, in Figure 7.2 there are two architecture models $am_1$ and $am_2$ conforming to two different architecture languages $ADL_1$ and $ADL_2$, and that $am_2$ has been generated by DUALLY starting from $am_1$ (by passing through an intermediate $m_{A0}$. Now, if two architects make concurrent changes to $am_1$ and $am_2$, the changed $am'_1$ and $am'_2$ must be checked (and possibly adapted) in order to re-establish their consistency.

![Figure 7.2: Concurrent changes in DUALLY](image)

In this future work it will be key to keep track of the various changes that have been performed in $am_1$ and $am_2$, this information can be stored in separate models ($c_1$ and $c_2$ in the figure). The $c_1$ and $c_2$ models may contain the difference between the original $am_1$ and $am_2$ models and the changed ones; in the current state of the art such a difference can be calculated automatically. Also, it is important to have tracing information of the execution of the two $ADL_12A0$ and $A02ADL_2$ transformations ($t_1$ and $t_2$ models in the figure). With these information available, a possible solution to this problem is to develop a generic mechanism that follows the steps sketched here:

1. for each change $c$ in $c_1$, identify which elements in $am_1$ have been impacted by $c$,
2. navigate the tracing information in $t_1$ and $t_2$ in order to identify which elements in $am_2$ correspond to the impacted elements in $am_1$,
3. for each change $c$ in $c_2$, identify which elements in $am_2$ have been impacted by $c$,
4. a generic component $CA$ applies the changes in $c_2$ back to $am'_1$ and the changes in $c_1$ back to $am'_2$. $CA$ contains a set of policies to solve possible conflicts that may arise when applying the changes.

Of course the sketched solution is somehow over-simplified, however it clearly shows that this problem is not specific to the software architecture community, it is rather more a pure MDE problem; also, it should be clear from the figure that the key element in the proposed solution are the $CA$ component and its solving policies. Future research work will be devoted to these parts of the solution, while I will reuse existing approaches for keeping tracing information from model transformation executions (e.g., the one proposed
in [82]), and for storing the changes made on the models under consideration (e.g., the one proposed in [23]).

In DUALLY, another generic issue regards techniques to check the quality of the transformations between ADLs. This issue may be studied from both a semantic and syntactic point of view. Addressing this issue at the semantics level will likely provide better results about the quality of the transformations. Concepts like semantic anchoring, semantic equivalence between models in MDE, model transformation testing will be considered in this part of research.

The current version of BYADL only allows the creation of new ADLs by either extending or customizing existing ones. The creation of a new ADL from scratch is not properly supported and investigated. In this case $A_0$ could play an interesting role by providing a minimal and generic ADL that can be extended as needed. I am planning also to further investigate the role of the BYADL model migrators. More precisely, the challenge is to investigate how to provide means to identify the parts of the software architecture that are affected by changes made by native ADL tools. In fact this enables software architects to understand if results of performed analysis remain valid even after changes or if the analysis must be re-performed.

Finally, the overall approach is applied to the software architecture domain and specifically to ADLs. However, I am investigating on the generality of the approach proposed in this dissertation, especially on its applicability outside the software architecture domain. If on one hand it is clear that MEGAFA is purely dedicated to the software architecture domain (since it is totally built on the ISO/IEC/IEEE 42010 standard) and thus it could be difficult to apply it in other domains, on the other hand I see the possibility for DUALLY and BYADL to be used outside the SA domain. More specifically, the DUALLY $A_0$ hierarchy (as presented in [37]) and its transformation management can be easily applied to different domains by simply substituting the $A_0$ metamodel with a different domain-specific metamodel. The definition of such a domain-specific metamodel is strategic and requires particular attention. Please refer to the discussion section in [45] for more details about the process we followed for defining $A_0$. We have demonstrated feasibility of this line of research in [28], in which we adapted $A_0$ to provide interoperability between component models in the field of real-time embedded systems. For what concerns BYADL, its composition operators and higher-order transformations are generic and agnostic of the contents of the metamodels being considered, thus they can be used as is in other fields. The only part that should be adapted, similarly to what said about DUALLY, is the one in which we use $A_0$ for helping the designer in composing the metamodels and for providing the semantics of the metamodels being composed.
REFERENCES

generation of architecture description languages. *Journal of Computer Science* 1,
4 (2005), 454–460.

Mellon, School of Computer Science, January 1997. Issued as CMU Technical
Report CMU-CS-97-144.


ing the UML

[6] Baumgart, A. A common meta-model for the interoperation of tools with heter-

4, 2 (May 2005), 171–188.

[8] Bézivin, J., Bouzitouna, S., Del Fabro, M., Gervais, M. P., Jouault, F.,
Kolovos, D., Kurtev, I., and Paige, R. F. A canonical scheme for model

[9] Bézivin, J., Böttner, F., Gogolla, M., Jouault, F., Kurtev, I., and
Lindow, A. Model Transformations? Transformation Models! In *Proc. 9th Int.
Conf. Model Driven Engineering Languages and Systems (MoDELS’2006)* (2006),

[10] Bézivin, J., and Gerbé, O. Towards a precise definition of the omg/mda frame-
work. In *Proceedings of the 16th IEEE international conference on Automated
software engineering* (Washington, DC, USA, 2001), ASE ’01, IEEE Computer
Society, pp. 273–.

in the Large and Modeling in the Small. In *Model Driven Architecture, Euro-
pean MDA Workshops: Foundations and Applications* (2004), vol. 3599 of LNCS,


REFERENCES


REFERENCES


